1 Algorithm

The following algorithm is implemented in Python in the back-end of the application, with full integration with the MySQL database. This final version of the algorithm has undergone multiple cycles of iteration, evaluated on a test data set (thirty generated playlists with labels representing user profiles) and validated by user feedback (on quality of matches for a group of twenty beta testers).

1.1 Creating a Feature Matrix

The step is to create a numerical representation of each user's musical tastes and preferences. To do so, create a feature matrix for each user. From the Spotify API, each user is associated with a list of their 100 top songs. Querying the API for song features gives a numerical vector representation for each song's 11 audio features (ex. energy, danceability, duration, etc.).

Note that each song is also tagged with a list of genre strings. Using text parsing with keywords (ex. "korea" for the K-pop genre), augment each song vector with binary indicator variables representing the 10 most popular genres. Now, each song has a numerical vector representation with length 21 (consisting of 11 fixed audio features and 10 extracted genre indicator variables). Combine the corresponding song vectors for each of the user's top 100 songs into a matrix $\mathbf{W}$, such that the feature matrix $\mathbf{W} \in \mathbb{R}^{21\text{x}100}$.

1.2 Reducing Complexity and Clustering

Next, the aim is to reduce the complexity of the data (100 songs per user) to a more manageable metric. In particular, this step reduces the user's feature matrix $\mathbf{W}$ to a set of five new feature vectors capturing different elements of the user's tastes. This reduction is done by taking the cluster centers after applying the K-means clustering algorithm for $K = 5$. This approach mirrors the way most people quantify their music preferences—as a mix of distinguishable styles and categories.

As a note, I experimented with reducing to a single vector that serves as the average of each of the columns in $\mathbf{W}$ with inferior results. Analysis showed that averaging the features for each song led to highly inaccurate results for listeners with diverse tastes. For example, users who only listened to sad ballads and heavy metal were represented by an average vector that resembled the features for pop music. In these cases, pairings are clearly inaccurate and reductive. The clustering approach generated significantly better matches in these instances due to the ability to capture multiple foci of music preferences, with $K = 5$ clusters empirically showing the best results.

1.3 Similarity Score

With each user now having a compact representation of music tastes (five vectors), it is possible to compute a similarity score between each pair of users. Visually, these similarity scores are stored in a matrix $\mathbf{S} \in \mathbb{R}^{\text{u x u}}$ where u is the number of users to be matched. Note that while $\mathbf{S}$ is symmetric, the duplicate values are not stored in the implementation for efficiency.

Apply the following procedure to calculate the similarity scores between each unique pair of users. Consider arbitrary users a and b, where a is represented by vectors $A = \{a_1, a_2, a_3, a_4, a_5\}$, and b is represented by vectors $B = \{b_1, b_2, b_3, b_4, b_5\}$. The similarity score between a and b, denoted $s_{(\text{a, b})}$, is calculated as $s_{(\text{a, b})} = \left(\sum_{i=1}^{5} \min_{b \in B}(|a_i - b|) + \min_{a \in A}(|b_i - a|) \right)$. Intuitively, this similarity awards a lower score if a and b have clusters that are closer, and a higher score if a and b have clusters that are very spread apart (representing distinct tastes). Using this metric, lower scores closer to 0 represent higher similarity (this score can be thought of as a numeric matching error between two users).

1.4 Greedy Pairing

With $\mathbf{S}$ fully populated, it now makes sense to create pairings based on the calculated scores. To pair users, follow a greedy algorithm. Take the pair of users with the lowest score in $\mathbf{S}$ (ex. user x and user y). Then, remove all candidate scores involving either x or y (conceptually removing the columns and rows representing x, y in $\mathbf{S}$). Continue until there are no matches remaining. The chosen

matches are the final set of matches minimizing the similarity loss and hence creating the highest quality matches.

1.5 Edge Cases

It is very possible for the total number of users to be odd, such that $u = 2n + 1$ where $n \in \mathbb{Z}^+$. In this case, proceed with the above algorithm to generate n matches, with one unmatched user c. For c, iterate through all u users to find the user d with the lowest score (an O($u - 1$) operation). Note that user d already has a match, so that after this adaptation user d will have two matches (the original match and now user c). Given the purposes of this algorithm to be integrated in a user-focused application, this solution is sufficient (as it is unlikely that a user will mind receiving two high-quality matches). This solves the issue of odd numbers by ensuring each user gets one match (and user d gets two matches), with user c getting as good of a match as possible from the given options.