Matching Users With Common Attributes

Deniz Cakiroglu

1 Introduction

This paper describes an algorithm for matching users based on a set of weighted attributes. The algorithm identifies pairs of users who share these attributes and assigns matches according to the degree of overlap. A practical application of this work is matching users by their most-listened artists and songs retrieved from the Spotify API.

An implementation of this algorithm is available here: https://github.com/DAB-Co/jam-server/blob/master/utils/algorithmEntryPoint.js.

2 Notation and Data Structures

Users: Let $U = \{1, 2, \dots\}$ be the set of users.

Preferences: Each user $i \in U$ has certain preferences, such as music artists or songs. Denote the set of all possible preference identifiers as $\mathcal{P}$ (e.g., unique IDs for songs).

Preference Weights: Each preference $p \in \mathcal{P}$ for user i is assigned a weight. We write $w = P_{i,p}$ to denote that user i has preference p with weight w in the database. If the preference is not in the database for user i, $P_{i,p}$ is undefined (or zero, depending on implementation).

New Preferences (Incoming Data): Suppose we receive a new list of preferences A for multiple users. For each user $i \in U$, let A_i be the set of tuples

$$A_i = \{(p, w) \mid \text{user } i \text{ is reported to have preference } p \text{ with weight } w\}.$$

Equivalently, $A_{i,p} = w$ means the new weight for preference p for user i in the incoming data.

Queue of Updates: We maintain a queue Q that lists all necessary updates to incorporate the new preference data A into the database P and the user graph G.

We define Q as a set of tuples (i, p, w) where $i \in U$, $p \in \mathcal{P}$, and w is the weight to be updated:

$$Q = \Big\{ (i, p, w) \;\Big|\; (p, w) \in A_i \;\wedge\; \big[(P_{i,p} \text{ is undefined}) \;\vee\; (P_{i,p} \neq w)\big] \Big\}.$$

In words, for each user i and each reported preference (p, w) in A_i, we add (i, p, w) to Q if (i, p) does not exist in P or if the stored weight $P_{i,p}$ differs from w.

Graph: Let $G = (U, E)$ be the undirected user graph, where U is the set of users and E is the set of edges. Each edge $(i, j) \in E$ is associated with a weight $G_{i,j}$, which denotes the total weight of common preferences for users i and j. We assume $G_{i,j} = G_{j,i}$. The size $|E|$ refers to the total number of edges in the graph (e.g., all common preferences among all users, representing them as connections).

Matches: Let $M \subseteq U \times U$ denote the set of user pairs that have been matched. In other words, $(i, j) \in M$ if and only if users i and j have matched before. By convention, matching is symmetric, so $(i, j) \in M \Leftrightarrow (j, i) \in M$.

3 Queuing Preference Changes

3.1 Algorithmic Description

When new preferences arrive, we examine each preference in A and compare it to the existing database P. If:

1. The preference does not exist at all for a particular user in P, or
2. The preference exists in P with a different weight,

we add an "update instruction (i, p, w)" to the queue Q; where $i \in U$, $p \in \mathcal{P}$, and w is the weight.

3.2 Implementation of Queuing Changes

An example implementation of this queuing process can be found in the function:

`https://github.com/DAB-Co/jam-server/blob/master/utils/algorithmEntryPoint.js#L46`

3.3 Complexity of Queuing Changes

Let $|A|$ be the number of users involved in the incoming data. For each user i, let $|A_i|$ be the number of preferences reported. If $\max_i(|A_i|)$ is the largest number of reported preferences for any single user, then constructing Q takes

$$O\Big(\max_i(|A_i|) \times |A|\Big).$$

We denote this size $|Q|$ (the total number of updates to be queued).

4 Applying Queued Changes to the Graph

Once Q is assembled, each preference update must be reflected in both the database P and the graph G.

4.1 Algorithmic Description

For each (i, p, w) in Q:

1. If p is new to $P_{i,\cdot}$, insert p for user i in P with weight w. If p already exists for i, we will adjust from the old weight $P_{i,p}$ to the new weight w.
2. Identify all users j who share preference p (i.e., $P_{j,p}$ is defined).
3. Subtract the old weight $P_{i,p}$ (if it existed) from the edge $G_{i,j}$, and then add the new weight w to $G_{i,j}$. Perform this symmetrically for $G_{j,i}$.
4. Update $P_{i,p}$ to the new weight w in the database.

4.2 Mathematical Formulation

Let $(i, p, w) \in Q$. Let w_{old} be $P_{i,p}$ if it is defined, or 0 if it is not defined. Let

$$\mathcal{U}(p) = \{\, j \in U \mid P_{j,p} \text{ is defined}\}$$

be the set of all users who have preference p in the database. Then:

1. If w_{old} is undefined, set $w_{\text{old}} = 0$ and add (i, p) to P.
2. For each $j \in \mathcal{U}(p)$ with $j \neq i$:

   $$G_{i,j} \leftarrow G_{i,j} - w_{\text{old}}, \quad G_{i,j} \leftarrow G_{i,j} + w, \quad G_{j,i} \leftarrow G_{j,i} - w_{\text{old}}, \quad G_{j,i} \leftarrow G_{j,i} + w.$$

   (Because the graph is undirected, $G_{i,j} = G_{j,i}$.)
3. Finally, set $P_{i,p} \leftarrow w$.

4.3 Implementation of Applying Changes

The implementation of this process can be found here:

https://github.com/DAB-Co/jam-server/blob/master/utils/algorithmEntryPoint.js#L234

4.4 Complexity of Applying Queued Changes

Let $|Q|$ be the number of updates in the queue. For each update $(i, p, w) \in Q$, the algorithm must apply changes to user i and any other users who share the preference p. In the worst-case scenario, every user shares preference p, so each update in Q could involve adjusting up to $|U|$ users. Consequently, the overall complexity is

$$O\big(|Q| \times |U|\big).$$

A practical illustration of this worst-case scenario arises when every user's preference weight changes simultaneously, while the underlying set of preferences (and thus the potential connections) remains the same.

5 Getting Matches

After updates, $G_{i,j}$ correctly reflects the total shared preference weight between users i and j. We then find matches by identifying, for each user, the neighbor with the largest edge weight.

5.1 Algorithmic Description

5.1.1 Selecting the Best Neighbor

For each user $i \in U$, define:

$$\mathcal{N}(i) = \{ j \in U : j \neq i \wedge G_{i,j} > 0 \}.$$

However this does not consider the fact that the user has been matched previously.

5.1.2 Previously Matched Neighbors

If we denote M_i as the set of users with whom i is already matched, then:

$$\text{Match}(i) \;=\; \arg \max_{j \in \mathcal{N}(i) \cap \overline{M_i}} G_{i,j}.$$

This excludes neighbors that user i has matched with in previous iterations.

5.1.3 No Unmatched Neighbors

Let $L \subset U$ be the set of users who cannot find any unmatched neighbor:

$$L = \{ i \in U : \mathcal{N}(i) \,\cap\, (\overline{M_i}) = \emptyset \}.$$

Users in L attempt to match with each other.

```
L_matched = set()
for u1 in L:
    if u1 in L_matched:
        continue
    found = False
    for u2 in L:
        if u1 != u2 and u2 not in L_matched and u2 not in M[u1]:
            L_matched.add(u1)
            L_matched.add(u2)
            found = True
            M[u1].add(u2)
            M[u2].add(u1)
            break
    if not found:
        # Lacking a suitable partner in L.
        # We could add u1 to LL, to try random matches later.
        LL.add(u1)
```

5.1.4 No matches left this iteration

Even after we attempt to match users in L with each other, we cannot guarantee that every user in L will find a partner. For instance, some users in L may already have matched with all available users in L or been ineligible due to external constraints.

Define

$$LL = \left\{ i \in L \;\middle|\; \nexists j \in L,\ (j \neq i) \land (\neg M_{i,j}) \right\}.$$

Hence, LL contains all users in L who could not be matched with any other user in L.

We then try to match each $i \in LL$ with a user $j \in U \setminus \{i\}$ for which $M_{i,j} = \text{false}$. Formally,

$$\forall i \in LL,\ \exists j \in U \setminus \{i\} : M_{i,j} = \text{false}.$$

Since we allow matching from the entire set U, it is possible for some users to have more than one match in this iteration.

5.1.5 Handling Multiple Matches from the LL Set

Even after attempting to match each user in LL with someone in G, multiple users from LL might target the same candidate. To limit over-matching, we impose a restriction that a user can be matched at most twice in a single iteration.

Why twice? If only one user remains unmatched in LL and all potential partners have already been matched exactly once, that user would remain unmatched without this rule. Thus, allowing up to two matches per user in this scenario ensures that everyone can potentially find a match.

```
LL_matched = set()
for u1 in LL:
    if u1 in LL_matched:
        continue

    # Try to match u1 with some user from the entire graph G
    for u2 in G:
        # Skip if already matched
        if M[u1][u2]:
            continue

        # Count how many matches u2 already has today
        m_count = 0
        if u2 in matched_today:
            m_count += 1
        if u2 in L_matched:
            m_count += 1
        if u2 in LL_matched:
```

```
        m_count += 1

    # If user u2 already has two matches, skip
    if m_count >= 2:
        continue

    # Otherwise, match them
    M[u1][u2] = True
    M[u2][u1] = True
    LL_matched.add(u1)
    LL_matched.add(u2)
    break
```

5.2 Implementation of Getting Matches

The complete implementation of the above steps is in:

https://github.com/DAB-Co/jam-server/blob/master/utils/algorithmEntryPoint.js#L55

5.3 Complexity of Getting Matches

- **Main matching step:** For each user, scanning potential neighbors can cost up to $O(|U|)$ if every user is connected to every other. Across all users, this naively is $O(|U|^2)$ in the worst case.
- **Handling L and LL:** In the worst case, all users end up in L, so matching within L (or later in LL) can also be $O(|U|^2)$.

Hence, the overall worst-case time is $O(|U|^2)$ for the matching phase.

6 Overall Complexity

Combining all steps:

$$O(|Q|) \quad + \quad O(|Q| \times |U|) \quad + \quad O(|U|^2)$$

is dominated by the largest term for large $|U|$. Consequently, we often consider $O(|U|^2)$ to be the leading term for big-data scenarios.

7 Additional Notes

Additional constraints (e.g., language compatibility, active/inactive users) are easy to incorporate. For example, if we store the set $LA_i \subset U$ of users who share a common language with i, then:

$$\text{Match}(i) \; = \; \arg \max_{j \in \mathcal{N}(i) \cap \overline{M_i} \cap LA_i} G_{i,j}.$$

Such constraints do not necessarily change the asymptotic complexity if membership checks (e.g., $j \in LA_i$) are $O(1)$ lookups. In practice, constraints can reduce the matching workload by shrinking the neighbor sets.