Cameron Chong
Kathryn Lecha
Emily Medema
Lauren St. Clair

# Recommender System Documentation

## 1. Problem Formulation

Our topic is to design an algorithm to recommend items based on the likes and dislikes of a user and their similarities to other users.

**Input**: Given a set of users $U = \{u_1, u_2, \ldots, u_i, \ldots, u_n\}$, Each user $U_i$ has a set of likes $L_i$ and dislikes $D_i$. Also, we are given a User $U_i$.

**Output:** A list of recommended items for User $U_i$ based on the likes and dislikes of Users similar to $U_i$.

We will calculate the similarity between users.

For a given user $U_1$, we will find the users similar to $U_1$.

Then, from the set of all items $X$, we will remove the items the given user $U_i$ has 'reviewed'.

$X - (L1 \cap D1)$, then we will recommend the items in similar to user $U_i$ list $L_i \in X$

## 2. Algorithm Design

### a. Brute Force

**i.** Compute the inversion array between a given user $U_1$ and every user $O(n^2)$ (brute force)

a. The score is based on the magnitude of difference between $U_1$ and other users.

b.

**Input:** Two ranked arrays A and B

```
numInv := 0
def compareInv(A, B):
    for i := 0 to n-1 do
        for j := 0 to n do
            if A[i] != B[j] and i != j then
                numInv := numInv + 1
    return numInv
```

ii. Create a similarity array between a given user U and every other user. A more similar user to you has fewer inversions.

**Input:** Given user U, list of Users of length n, each user has m liked items and o disliked items

```
User U
simArray
for i := 1 to n-1 :
     Sim = compareInv(U, i)
     simArray.append(sim)
```

iii. Find items that users with high similarity to the given user have reviewed. Retrieve all items and remove all items that you, the given user, have reviewed

**Input**: A similarity array S, Matrix M(n users by m items), given User $U_i$

```
S.sort(descending)
Take the first w users.
list_items = []
for user in S:
     items = M[user][:]
     for item in items:
          if(item not in list_items and item not
          in M[U][:]):
               list_items.append(item)
 return list_items
```

iv. Return that list of reviewed items

As you can see the brute force of our algorithm has the time complexity O(nmo). However, m and o are equal therefore it is $O(nm^2)$

*Proof of Correctness for Brute Force:*

Proof by Induction:

Prove that given a user U and a set of Users M of length m, each with a set of likes and dislikes a recommendation can be made to the given user U.

**Invariant**: After going through the algorithm, a list of recommended items will be provided to the given user.

**Base Case**: m = 1

If m = 1 that means that there is one user other than the given user.

For each user (only 1) we will call simCompare(U, M[m]) and the similarity value will be returned.

Then, a loop will run comparing the rankings of M[m] to User U's ranking. The returned value will be added to a series.

This series will be sorted so that the most similar users are first.

The algorithm will then loop through the users (in order of

similarity). If an item in M[m]'s likes is not in the recommended_items, U's likes and U's dislikes it is added to recommended_items.

Therefore, the base case returns a list of items User U has not seen before based on the similarity of M[m]

**Inductive Case:** Assume m = k is true, prove that for m = k+1, the loop invariant holds.

If m = k+1 that means that there are k+1 other users than the given user. However, the primary functionality is the same.

For each user we will call simCompare(U, M[m]) and the similarity value will be returned.

Then, a loop will run comparing the rankings of M[m] to User U's ranking. The returned value will be added to a series.

This series will be sorted so that the most similar users are first.

The algorithm will then loop through the users (in order of similarity). If an item in M[m]'s likes is not in the recommended_items, U's likes and U's dislikes it is added to recommended_items.

Therefore, the base case returns a list of items User U has not seen before based on the similarity of M[m]

Thus, we can see that the brute force algorithm will work for all cases if users. QED.

**b. Improved Algorithm**

An easy way to improve the run-time of the algorithm would be to change the method of counting inversions. This way it would be O(nlogn) instead of $O(n^2)$.

However, this is really hard to adapt for our ranking and two arrays. Therefore, we decided to switch to calculating similarity via the Jaccard Index. This follows the aforementioned equation :

$$s(U1, U2) = (|L1 \cap L2| + |D1 \cap D2| - |L1 \cap D2| - |L2 \cap D1|) \div |L1 \cup L2 \cup D1 \cup D2|$$

This will also allow us to use sets, which are much faster than matrices and arrays. For example, a set will let us check if an item is already in a set in O(1) time rather than iterating through an array.

Here is the pseudocode for our improved algorithm:

*Similarity Calculation Method*

```
def sim(L1, L2, D1, D2):
    num = 0;
    dividend = length((L1 intersection (L2)))
     + length((D1 intersection(D2)))
     - length((L1 intersection(D2)))
     - length((L2 intersection(D1)))
    divisor = length((L1 union L2 union D1 union D2))
    num = dividend/divisor
    return num
```

*Loop to Calculate Similarity to Given User*

```
Set of given_user_likes
Set of given_user_dislikes

Empty series sim_series of length n
for i in users:
        Set of i likes user_l
        Set of i dislikes user_d
        num = sim(given_user_likes, user_l,
given_user_dislikes, user_d)
        Add num to sim_series
```

*Loop to Recommend Items to Given User*

```
Sort sim_series
list_items = []
for user in sim_series:
    Set of items user likes: user_items
    for item in user_items:
        if item not in list_items
           and item not in given_user_likes
           and item not in given_user_dislikes:
            list_items.append(item)
Return list_items
```

This code runs in O(nm) time.

*Proof of Improve Algorithm Correctness*

**Proof of Loop to Calculate Similarity to Given User:**

**Invariant:** At the end of an iteration the **sim_series** contains the similarity index of the first i users and the current i user. The loop starts at user 0 and ends at user n (where n number of users).

**Proof by Induction:**
*Base Case*: n = 1
According to the loop invariant, after going through the loop 1 time, **sim_series** should be equal to the similarity of user1 and the given user.

Variables user_l and user_d become the sets of $user_i$'s likes and dislikes.

Then the **Similarity Calculation Method** is called. A similarity index is returned and saved into **sim_series**.

Now the loop is complete. Our invariant is true, therefore the correctness of this loop is true.

*Inductive Step*: Assume that the invariant is true for n = k. Prove that the invariant is true after going through the loop k+1 times.

At k+1, **sim_series** holds the similarity index of the previous k users and user_l and user_d will become k+1's sets of likes and dislikes.

Then the **Similarity Calculation Method** is called. A similarity index is returned and saved into **sim_series**.

Now the loop is complete. Our invariant is true as **sim_series** contains the previous k user similarity indices and k+1's, therefore the correctness of this loop is true.

**Proof of Loop to Recommend Items to Given User:**

**Invariant**: At the end of an iteration, **list_items** contains a list of items from similar users likes that were not in the given_users likes or dislikes. The loop starts at user 0 and ends at user n (where n is the number of users).

**Proof by Induction:**
*Base Case:* n = 1
According to the loop invariant, after going through the loop 1 time, **list_items** should be equal to the liked items of user1 that are not disliked or already liked by the given user.

**User_items** is equal to the items that user1 likes.

A second loop iterates over the items that the user likes.

The **invariant** of this loop is the same as the invariant for

the outer loop, except it iterates from item 1 to item *m* of the user's liked items.

In this loop, the item is checked to see if it is already in **list_items**, if the given user has disliked it, or if the given user has already liked it. If none of these are true the item is added to **list_items**.

The loop is now complete. At the end of the iteration, **list_items** contains the liked items of the user that are not already in **list_items** and that the given user has not already liked or disliked. Thus, the invariant holds.

*Inductive Step:* Assume that the invariant is true for n = k. Prove that the invariant is true after going through the loop k+1 times.

At k+1, **list_items** should be equal to the liked items of the previous k users that are not disliked or already liked by the given user.

**User_items** is equal to the items that k+1 likes.

Then a second loop starts iterating through the items that k+1 likes.

The **invariant** of this loop is the same as the invariant for the outer loop, except it iterates from item 1 to item m of the k+1's liked items.

In this loop, the item is checked to see if it is already in **list_items**, if the given user has disliked it, or if the given user has already liked it. If none of these are true the item is added to **list_items**.

The loop is now complete. At the end of the iteration, **list_items** contains the liked items of k+1 that are not already in **list_items** and that the given user has not already liked or disliked as well as of the previous k users. Thus, the invariant holds.

**Proof of the Similarity Calculation Method**

**Invariant:** Provided the sets of L1, L2, D2, D1 a Jaccard similarity index exists and will be returned.

Proof by example:

L1 = {1, 2, 3}
L2 = {2, 3, 5}

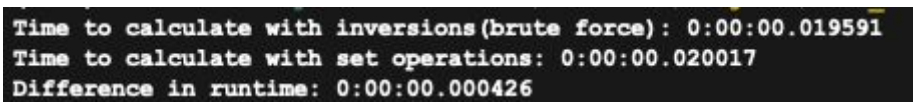
```
D1 = {5, 6, 7}
D2 = {6, 8, 9}

length((L1 intersection (L2)))
      + length((D1 intersection(D2)))
      - length((L1 intersection(D2)))
      - length((L2 intersection(D1))) = 2

length((L1 union L2 union D1 union D2)) = 8

Num = 2/8 = 0.25

Method returns an index of 0.25. This means that the two
users are 0.25 similar.

As you can see a result exists and the loop invariants hold
thus, the algorithm is correct.

QED.
```

## 3. Implementation and Empirical Evaluation

Originally we were using the MovieLens dataset, however, it was very hard to work with as the brute-force method took hours to run. Therefore, we created our own dataset which is much smaller. Our implementations would still work for any dataset (if you have the time to run it), provided you format the data in the correct way from the dataset.

The implementation of the algorithm can be found on our Github. The file setup has two feature branches for determining each algorithm, where each algorithm exists in its own directory. In the main directory is a python script that has four methods, one for each of the two different types of similarity calculations, and one for each actual algorithm. Running this script will display the results on sample data for each algorithm as well as the difference in runtime.

It should be noted that the "Algorithm B", using set operations, is listed as running in almost the same time or slower than counting inversions. However, as set operations scale faster than matrix, as the input grows asymptotically, the set operations method will operate faster than the matrix, according to their big O notations discussed in the next section.




One major concern in implementation was the time that it takes to run and to implement. Because of this, we shrunk the dataset, so implementation testing could be

done in a timely manner. Each member of the group had a large number of other projects and responsibilities to address in the same timeframe, so development of an algorithm was stunted by that. We discussed the required output and determined the best one for us accordingly. This will be discussed in more detail in the next section.

## 4. Data Structures

We then thought about what data structures we should use. Currently, we are using a dataframe (which is similar to a matrix). This is not the best data structure for us as we are iterating through the matrix in order to compare users and find recommended items. If we pick a better data structure we could greatly improve the runtime. However, since a dataframe is indexed, it can also act as a dictionary (hashmap) of lists. In this way, the runtime is greatly improved over the matrix, as the indices created faster lookup times. Both implementations use dataframes.

One data structure we could use is graphs. Since we are focused on user-item relationships, if we initialized a graph G(V, E) where each node is either an item or a user. When a user 'likes' an item a directed edge is formed with a weight representing the rank of the item. When two users are compared an bi-directed edge is formed with the weight representing the similarity.

However, the way that our data is presented is that each user has a ranking of movie items. This is hard to represent in a graph. According to this source, a tripartite graph would be more efficient, however, it does not work with the problem formulation and data that we have.

Another possible data structure we considered was priority queues. If we stored the users and sorted by similarity we could avoid sorting, which would reduce complexity. However, this would still make us not able to use the aforementioned hash-table method, which would create more complexity comparatively.

In the end, we decided to target the step causing the most complexity outside of the inversions and reduce it. Therefore, we decided to implement a hash-table (or similar) so that we would not have to iterate over the given user's items every time. This would reduce that complexity to O(nm).

Also we considered sets, which are much faster than matrices and arrays. For example, a set will let us check if an item is already in a set in O(1) time rather than iterating through an array.

We eventually decided to use a combination of sets and series (similar to hash tables) reducing our complexity down to O(nm).