Assignment 2

Experimental Analysis of Prediction Accuracy achieved by Recommender System Algorithms

The algorithm used for evaluating prediction accuracy was N-fold cross validation (more specifically, the Leave One Out strategy) offline with synthetic datasets. Other algorithms I used to compare were the item-based and user-based recommendation strategies, with adjustments for top-K and threshold-based sorting for nearest neighbours, and checking differences when negative similarities were included/excluded.

My code has 5 major functions:

- `readFileFromPath` takes in a string to the path of the file, opens the file, and returns the data inside it.
- `parseInput` takes this string and processes it so that it is broken down and stored into the variables.
- `simOf` takes changes the similarities affected by the change in rating of the given parameters in the similarity matrix.
- `pred` takes two parameters (user, item) and makes a prediction rating for them based on the similarity matrix.
- `leaveOneOut` is the function that performs the strategy mentioned above and calculates the MAE for prediction accuracy.

The important variables of note here are:

- `u: [user (string)]` – array of all users
- `p: [item (string)]` – array of all products/items
- `r: { user (string): { item (string): rating (Number) } }` – ratings Object
- `u_sum: { user (string): ratings_sum (Number) }` – stores sum of user ratings
- `u_num: { user (string): ratings_num (Number) }` – stores number of items rated by user
- `sim: { item1 (string) : { item2 (string): similarity (Number) } }` – matrix of similarity scores

Runtime for implementations was minimized by using data structures that allow quick access (such as the JavaScript Object and arrays). Maps were also considered but performed similar to Objects in this scenario. Sum of ratings by a user are stored in u_sum and number of items rated by a user is stored in u_num for quicker calculation (and re-calculation) of averages. Looping through arrays was done by using for loops that walked through the index values of the array (arr)

```
for (let index = 0; index < arr.length; index++) { ... }
```

instead of using a for-of loop

```
for (let elem of arr) { ... }
```

which cut down a significant amount of computation time. Similarities are only recomputed for the item to be predicted instead of recomputing the entire similarity matrix every time a rating is crossed out.

For the leave-one out cross validation strategy, I looped through the dataset and “crossed out” or set the current rating to no rating (value of 0). Then, the prediction function for the recommender system was used to predict the rating that was crossed out. Following this, the Mean Absolute Error (MAE) was calculated to check the prediction accuracy. The crossed-out rating is reset to its previous value and the loop continues onto the next rating to be crossed out. Ratings that are already 0 (unrated) are skipped from the prediction cycle.

When comparing the user-based and item-based nearest neighbour recommendations, I used the dataset in the assignment2-data.txt file provided. Using the top-K strategy with a neighbourhood size of 5, I found that item-based strategy gave a higher MAE of 0.7455 as compared to user-based which had an MAE of 0.6840. User-based calculations also tended to be faster by a small margin. Over an average of 3 runs, item-based took about 33,769 ms to run as compared to 30,213.33 ms for user-based.

Threshold-based tended to yield more accurate results as compared to top-K selection when the execution time was considered. Using item-based strategy, ignoring negative similarities, when a neighbourhood size of 50 was used, the MAE turned out to be 0.6639 calculated in 69,143 ms as compared to when a threshold of 0.1 was used and the MAE was 0.6639, calculated in 29,650 ms. More experiments were performed with different values for neighbourhood size and threshold.

Neighbourhood size	MAE	Time taken	Threshold	MAE	Time taken
2	0.8190	29,309 ms	0	0.663939	30,441 ms
5	0.7455	33,075 ms	0.05	0.663929	30,631 ms
10	0.7133	36,109 ms	0.1	0.663920	30,970 ms
20	0.6915	45,327 ms	0.25	0.665251	31,313 ms
50	0.6738	69,230 ms	0.5	0.675314	29,639 ms
100	0.6676	124,219 ms	0.75	0.707434	29,567 ms

- Figures above are for item-based, ignoring negative similarities

Neighbourhood size	MAE	Time taken	Threshold	MAE	Time taken
2	0.7508	32,030 ms	0	0.662945	30,937 ms
5	0.6840	33,544 ms	0.05	0.663740	31,561 ms
10	0.6659	33,833 ms	0.1	0.663504	29,857 ms
20	0.6628	34,273 ms	0.25	0.675389	29,996 ms
50	0.6629	35,612 ms	0.5	0.733840	31,036 ms
100	0.6629	40,125 ms	0.75	0.738286	30,840 ms

- Figures above are for user-based, ignoring negative similarities

Looking at the tables above, we see multiple trends. As neighbourhood size increases, MAE decreases (meaning that accuracy increases with larger neighbourhood sizes) but plateaus at around 50. We also notice that the time taken by the algorithm keeps increasing consistently, which suggests that increasing neighbourhood sizes, boosts accuracy but with a performance cost. On the other hand, the lower the threshold value, the lower the MAE (which means higher accuracy). There is no relation between the threshold value and performance since there is no sorting involved. Overall, it looks like the most accurate results would be produced when a threshold value of 0 is used.

Ignoring negative similarities yielded a lower MAE (0.7455) than when considering negative similarities (0.8052) into the calculation, when all other parameters are kept same (item-based,

top-K, neighbourhood size = 5). This means that ignoring negative similarities yields more accurate results as compared to when negative correlations are included.

As mentioned in the tables above, each prediction usually takes about 30 – 45 seconds to execute. Threshold based tends to run faster than top-K since there is no sorting involved. The implementation with user-based algorithm tends to have a lower MAE and run faster than the item-based implementation. These trends are likely to be based on the way the algorithms are implemented.

While user-based recommendations turn out to be more accurate, it works for small datasets and causes efficiency problems in larger datasets. Thus, we need to consider item-based recommendations since we can pre-process item similarity and it tends to be more stable as compared to user similarity. For a real-time online movie recommendation system, we can start out by precomputing the entire similarity matrix once and then only modifying the similarities corresponding to the changes in ratings. The underlying algorithm could use item-based recommendations, ignoring negative similarities and using a certain threshold value for selection. In a real-world system, there would be a need to update this threshold depending on how popular a movie is overall, and how the ratings may be inflated in that case. There is also a need to consider other factors such as similarity based on genres and cast, “dislikes” or “favorites” of a user, recommendations that the user asks the system not to recommend again, etc.

More reviews improve the prediction accuracy, but after a certain point the accuracy stabilizes with diminishing returns. If there are a lot of users with very few reviews, we would be faced with the sparsity problem where we have a ratings matrix that has many missing ratings. To

solve this, we can rely on other factors mentioned above, and use a graph-based approach as discussed in class (applying associative retrieval techniques) to reduce the sparsity problem. On the other hand, users with a lot of ratings would make our system run slower, and in such a case we would need to change our threshold and make it higher, or alternatively switch to a top-K approach to improve performance at the cost of accuracy.