DS-GA 1004 Big Data Capstone Project Final Report

Ceci Chen, Lia Wang, Maggie Xu

Link to GitHub repository: https://github.com/nyu-big-data/capstone-project-g-38.git

For our project, our group initially adopted an individual approach, where each member worked on the first part of the project independently. We then compared our individual efforts to select the best version. For the second part of the project, we collaborated closely, experimenting with different software, clusters, and platforms to determine the most effective solution. Specifically, we executed our code on Dataproc, Greene Cluster, and Google Colab to find the optimal environment. For the first two questions, we did not manage to run the full set on any other platforms until we successfully ran the full dataset on Colab. We used the GPU parallel processing accelerator and leveraged efficient data manipulation libraries using libraries CuPy and RAPIDS (cuDF) to significantly speed up the operations as shown below

```
!wget -nc https://raw.githubusercontent.com/rapidsai/rapidsai-csp-utils/main/colab/rapids-colab.sh
!chmod +x rapids-colab.sh
!./rapids-colab.sh stable
import cupy as cp
import cudf
```

For the remaining questions, we managed to run our code on both Dataproc and the local cluster using Pandas and Spark, ensuring thorough testing and optimization.

Customer Segmentation:

1. To address the task of finding the top 100 pairs of users with the most similar movie-watching styles, using a MinHash-based approach allows for an efficient approximation of Jaccard similarity between sets of movies rated by each user. The first step we did is to preprocess the data. We utilized the raw full ‘rating’ datafile to create a dictionary where each key is a user ID and the corresponding value is a set of movie IDs that the user has rated. This set represents the 'movie-watching style' of the user, and then achieves this by grouping the DataFrame by 'userId' and collecting 'movieId' into sets. Once the data is structured, the next step we did is to initialize and populate MinHash objects for each user. MinHash is a probabilistic data structure that

efficiently estimates the similarity of datasets. In this case, each user's movie set is used to update a MinHash object. These MinHashes are then inserted into an LSH index which allows for querying similar items quickly. With the LSH index populated, we queried for each user to find other users whose MinHash signatures are similar, based on the LSH threshold. The query will return a list of user IDs who are potential "movie twins". This list excludes the user itself to avoid self-matching. The results are collected into pairs, ensuring each pair is unique and no duplicates are included. Finally, to determine the top 100 pairs, we sort these unique pairs by the Jaccard similarity, estimated using the Jaccard method available on MinHash objects. The pairs with the highest 100 similarity are then selected(result output attached). By looking at the top 100 pairs similarity dataframe, we noticed that the dataset contains all records of user pairs with a Jaccard similarity of 1.0, indicating that these pairs share identical items rated. The broad distribution of users and the lack of repeat pairings beyond two occurrences per user suggest a diverse and large item base or user interaction style that results in many unique one-time high-similarity interactions. This could imply that while Jaccard similarity effectively identifies users with identical interaction patterns, it may overlook the depth of these interactions, such as the number of ratings per item, which could provide more nuanced insights into user behavior.

List of top 100 most similar pairs

jaccard_similarity	user1	user2	jaccard_similarity	user1	user2	jaccard_similarity	user1	user2
1.0	72276	111612	**1.0**	227999	236285	**1.0**	207621	244088
1.0	35628	169217	**1.0**	4617	237043	**1.0**	43885	138202
1.0	14443	18603	**1.0**	25208	294383	**1.0**	49090	92192
1.0	234323	264008	**1.0**	193242	204748	**1.0**	128516	196418
1.0	27003	299432	**1.0**	96331	308385	**1.0**	240268	317002
1.0	113505	326723	**1.0**	31964	300506	**1.0**	98519	156728
1.0	76286	99734	**1.0**	68330	201366	**1.0**	21112	297914
1.0	68757	92370	**1.0**	98843	206344	**1.0**	22384	120993
1.0	92396	208026	**1.0**	31140	175306	**1.0**	138087	202918
1.0	138203	312682	**1.0**	51319	80485	**1.0**	189439	197122
1.0	9002	255504	**1.0**	294858	307859	**1.0**	121017	298050
1.0	109192	289236	**1.0**	112363	159721	**1.0**	103350	264011

1.0	143375	297542	**1.0**	133168	189687	**1.0**	104991	276172
1.0	147482	325411	**1.0**	126686	312243	**1.0**	54093	303344
1.0	268449	313635	**1.0**	169545	238088	**1.0**	145156	168847
1.0	202089	270838	**1.0**	235808	293733	**1.0**	188776	201344
1.0	76913	261526	**1.0**	141674	278548	**1.0**	269807	287468
1.0	27735	103004	**1.0**	129847	158783	**1.0**	280839	321485
1.0	5291	318135	**1.0**	27735	68330	**1.0**	44190	135138
1.0	11341	172163	**1.0**	231837	245162	**1.0**	37106	301120
1.0	16620	242856	**1.0**	188272	211516			
1.0	91288	113374	**1.0**	234590	258916			
1.0	159009	240511	**1.0**	38553	124779			
1.0	13580	308567	**1.0**	3649	65653			
1.0	75901	316199	**1.0**	9195	327267			
1.0	63935	108808	**1.0**	129993	154447			
1.0	218646	273592	**1.0**	238435	279885			
1.0	85416	204748	**1.0**	113374	289236			
1.0	238	300592	**1.0**	119491	175306			
1.0	159009	314983	**1.0**	24327	136170			
1.0	76286	174344	**1.0**	40794	59672			
1.0	2756	119818	**1.0**	105353	320419			
1.0	165526	289845	**1.0**	140077	305993			
1.0	45703	276732	**1.0**	31299	208797			
1.0	37980	101286	**1.0**	35859	238192			
1.0	157570	217479	**1.0**	95365	146655			
1.0	74979	290838	**1.0**	37106	268763			
1.0	56408	264852	**1.0**	8956	248949			
1.0	67915	228825	**1.0**	32723	72973			
1.0	228099	232515	**1.0**	32838	277891			

2. To validate our results from question 1 and checking the effectiveness of using MinHash to find user pairs with similar movie watching styles, we compare the average correlation of numerical ratings between the top 100 similar user pairs identified by MinHash against 100 randomly

selected user pairs from the full ‘rating’ dataset. We first define a function ‘calculate_average_correlation’ that takes a list of user pairs and the ratings DataFrame as input, and calculates the average Pearson correlation coefficient of the numerical ratings for movies both users in each pair have rated. The function handles the missing data by filtering out the NaN pearson correlation to avoid skewing the average. The next step we did is to calculate correlation for top 100 and random pairs. We utilize the ‘calculate_average_correlation’ function to compute the average correlation for the top 100 pairs identified from the MinHash approach. And then, generate 100 random pairs from the list of all possible user pairs in the full dataset by random sampling and calculate their average correlation. The results we got for average correlation are: Average correlation of top 100 similar pairs: -0.023626062819918664, Average correlation of 100 random pairs: 0.11596791586208896. Some of the factors that could affect this negative results in the correlation might be the focus of the MinHash, since it is generally used for estimating the Jaccard similarity between sets, which in this case are the sets of movies that users have rated. If MinHash effectively identified users who watched similar movies but did not consider the actual ratings given to these movies, the correlation in their rating patterns might not necessarily be positive.

Movie Recommendation:

3. We first did a simple train set, test set and validation set split. This is due to the reason that the popularity model is a non-personalized model. The popularity model does not require specific user data patterns to produce the recommendations. As a result, the same set of items is recommended to all users as the recommendations are based on global popularity rather than individual preferences. For the code specifically, we first shuffled the data to ensure that the data splitting does not include any bias from the order of ratings. We split the data set into 80% training set, 10% testing set, and 10% validation set. The same size of the testing and validation sets ensures the balance between model evaluation and fine-tuning.

   For the latent factor model, to handle the dependencies inherent in recommendation systems, we define a function to conduct the data split. In the function, we utilize Spark's DataFrame operations, including ‘orderBy’ with a random seed for shuffling, and ‘limit’ to partition the data. The splitting process ensures 80% of the ratings are used for training, and the remaining are evenly split between validation and test sets. We made sure the code iterates over

all unique users in the dataset, and then we combine the resulting sets into unified DataFrames using ‘union’.

4. To construct the popularity based model, we started our preprocessing by first grouping the training set by “movieId” to count the number of each movie received in the set. Then, we sorted the movies in descending order based on their “ratingCount” to determine the top 100 most popular movies and finished constructing the popularity baseline model. To further evaluate the model, we aggregated the actual movies rated by each user in the test set into a list and assigned the same list of top 100 popular movies to each user as their predicted recommendations. By looping three ranking metrics (Precision at K, Mean Average Precision, and NDCG at K) over users, we are able to evaluate how well our popularity baseline model works. For Precision at K here, we assign K = 100 and this model gives us a final score of 0.0201, which means that about 2.01% on average are movies that the user has actually rated out of the top 100 recommended movies. For Mean Average Precision, it gives a final score of 0.078, which means that the average precision of the relevant movies across all users in the recommendation list is about 7.82%. For Mean NDCG at 100, the final score is 0.221 and this suggests that the ranking quality of the recommended movies is about 22.14% of the ideal ranking, which may mean that the recommended movies are not very well-ordered in terms of relevance to the users. From all three scores above, we can conclude that although popularity baseline mode can indicate some relevant movies, but its model accuracy is still strongly restricted by its non-personalized characteristics.

5. We train the recommender system using Apache Spark’s Alternating Least Squares method, and we initiate the parameter grid to conduct cross validation for the parameters. Specifically, we included the **rank (dimension) of the latent factors, the number of iterations, and the regularization parameter, with the best model result: rank = 10, maxIter = 5, and regParam = 0.1**. Using these results, we trained the model using the train set and evaluated it on the validation set.

   The RMSE (0.814) is calculated on both validation and test sets to evaluate the model performance. To further assess the model’s performance, we defined three metrics (Precision at K, Mean Average Precision, and NDCG at K). We first extracts the top 100 recommendations for each user from the trained model, maps these predictions to actual movie ratings from the

validation set, and computes the evaluation metrics across all users. These metrics are calculated using the PySpark recommendation system package. The mean precision at K (where K=100) is 1.0558e−5, which indicates that very few of the top 100 recommended movies for each user were actually rated by them in the validation set. This suggests that the recommendation system might not be accurately capturing user preferences or that the top recommendations are not aligning well with users' interests.The mean MAP is 2.1570e-5. Similar to precision, the low MAP indicates that the average precision of the recommendations across all users is very low, meaning that the relevant items are not frequently being ranked highly. The mean NDCG at K is 0.000173, which signifies that even when the system does recommend relevant items, they are not placed high in the recommendation list.

However, when we ran the code using the small dataset, the metrics were a lot higher as following:

	Small Dataset	Full Dataset
Mean Precision at 100	0.006672240802675586	1.0558e−5
Mean Average Precision	0.020343701834365197	2.1570e-5
Mean NDCG at 100	0.09531255666403379	0.000173