Big Data Capstone - Group 20

By Abhinav Krishnan, Jason Moon, Raj Raman
https://GitHub.com/nyu-big-data/capstone-project-cap20

Q1. List of top 100 most similar pairs sorted by similarity

The top 100 most similar pairs are listed in the “top_100_filtered5.csv” in GitHub.

We use the CountVectorizer function from PySpark to convert the movieIds into a sparse vector. Then, we apply the MinHashLSH with 5 Hash Tables followed by the approxSimilariyJoin to compute the Jaccard Distance. We use Jaccard Distance as our similarity metric, where the closer it is to 0 the more similar the users watching styles are. This is used instead of Jaccard similarity because the MinHashLSH function from PySpark only implements the distance metric. Jaccard Similarity is simply = (1 - Jaccard Distance), so our analysis does not change.

Notably, when we include all users, the top 100 Jaccard distances were all 0. When we investigated this, we found that this was driven by user pairs who had watched the single same movie. Hence, we filtered users who rated at least 5 movies. Assuming all movies are randomly watched with equal probability, the chance of an exact match of 5 movies between 2 users is extremely small. However, the top 100 Jaccard distances were still all 0, and these were driven by user pairs who watched the same set of movies. We do not further filter out the 0s since this indicates these 5 movies are significantly more popular than a random pick would imply, and there is value in identifying these movie twins.

Q2. Comparison between the average pairwise correlations between ratings of the "movie twins" and randomly picked pairs.

To compute the correlation of the 100 movie twins, we joined the ratings table with the top 100 table from Q1. For each pair, we only keep the ratings of movies that both users have watched. Then, we used the corr function from PySpark to compute the correlation between the movie ratings of each pair and average the correlations over the 100 pairs.

For the random pairs, we select a random sample of 200 userIds from the ratings dataset and pair them up. We apply the same filtering and correlation function for each pair and average over 100 pairs.

We found that some pairs have nan correlation since they have the same rating for all the common movies (i.e. no variance). We decided to ignore these instances from the average correlation calculation.

	Average Correlation
Movie Twins	0.1276
Random Pairs	0.1352

The results above contradict what we would have expected - that the correlation of the movie twins should be higher than random pairs. However, this does not need to be the case since the “movie twins” are chosen based on the movies they have watched but not based on the ratings similarity. Hence, two users can watch the same movies but rate them very differently causing the correlation to be low. The random pairs are high by chance, since, on another run, we got a very low correlation of 0.03.

Q3. Train / validation split generation

Our train, validation, and test sets were generated in the following way:

1. We collect unique users
2. We **randomly split** all **users** into 70% train users, 15% validation users, and 15% test users.
3. For each train, validation, and test user, we collect their movies and ratings
4. For each **validation user** and **test user,** we **split** their **ratings randomly** into two equal sized sets. The **first 50%** of ratings are preserved **for the validation/testing**. The other **50%** of ratings are **joined with our Train set** to ensure that there is no cold start problem (i.e. users who have no ratings in the train set) when we evaluate the model on the validation and test sets.
5. The final train set has all users and all movies.
6. The final validation set is the first 50% set of the validation user ratings, and the final test set is the first 50% set of the test user ratings.

Additional pre-processing of the data

1. In our recommender system datasets, we dropped users with a ratings count < 10 to make sure all users have rated enough movies to provide a recommendation and the evaluation can be accurate.

2. In order to calculate the Ranking metrics in Spark's MlLib Library, we post-processed our resulting dataframes to convert them into RDDs with recommendations and ground truth labels as key, value pairs.

Q4. Evaluation of popularity baseline

The popularity baseline was built and evaluated in the following way:

1. We calculate a **damped popularity** score for each movie with a damping factor of 1000. This was tuned manually until it generated reasonable predictions.
2. In the test data, for each user A, we **recommend** the top 100 movies in descending order of damped popularity. We ensure that user A is only recommended movies that have ratings in the test data. This is to ensure fair evaluation against the available ground truth. In some cases, the user might not have 100 ratings in the test set, in which case they get recommended as many movies from the popularity baseline as they have ratings for in the test set. We considered making a smaller prediction set, but we decided to stick with 100 to match the instructions provided in the README.
3. In order to generate the **ground truth labels** of positive interactions, we filter each user A's ratings based on the mean rating for user A. If the rating for a movie is above the mean rating for user A, that movie is considered to be a positive interaction.
4. We calculate the following metrics for our **top 100 recommendations against the ground truth labels** of positive interactions.

MAP - Mean average precision (MAP) for a set of queries is the mean of the average precision scores for each query. Average Precision.

NDCG - Normalized Discounted Cumulative Gain (NDCG) is the Discounted Cumulative Gain (DCG) that is normalized by dividing by the ideal DCG (IDCG). The DCG uses a graded relevance scale of documents from the result set to evaluate the usefulness, or gain, of a document based on its position in the result list.

Recall - Recall is the fraction of the documents that are relevant to the query that are successfully retrieved. We report this to analyze how our model makes the trade-off between making precise recommendations and capturing all of a user's possible preferences.

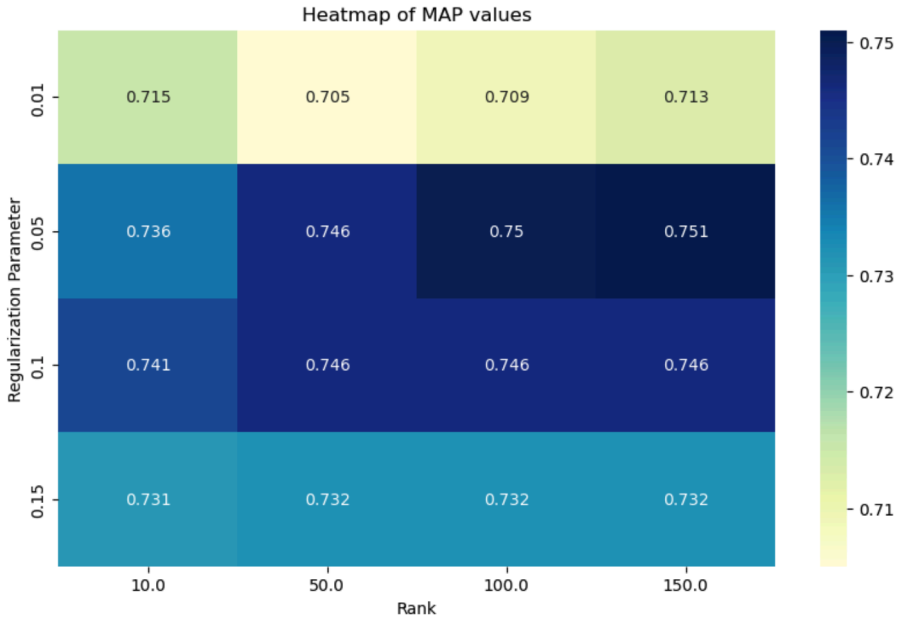
MAP	NDCG at 5	NDCG at 100	Recall at 5	Recall at 100
0.684	0.744	0.849	0.382	0.946

We find that our Damped Popularity model is a strong baseline that already does a commendable job at making recommendations.

Q5. Latent factor model's hyper-parameters and validation

To get the ground truth labels of positive interactions, we apply the mean-based method from the Popularity Model. To generate the Top 100 recommendations, we sorted the predicted ratings and calculated the rank of the movies. We got the top 100 ranked movies as our recommended movies.
Our objective is to tune the *rank* and *regParam* hyperparameters of the ALS. We did a grid search to find the best hyperparameters that achieve the highest Mean Average Precision on the validation set. We try values of *rank*={10, 50, 100, 150}, and *regParam*={0.01, 0.05, 0.1, 0.15}. We summarize the results in the table below and visualize them using a heatmap.

Rank	regParam	MAP
10	0.01	0.715
	0.05	0.736
	0.1	0.741
	0.15	0.731
50	0.01	0.705
	0.05	0.746
	0.1	0.746
	0.15	0.732
100	0.01	0.709
	0.05	0.750
	0.1	0.746
	0.15	0.732
150	0.01	0.713
	0.05	**0.751**
	0.1	0.746
	0.15	0.732



The best hyperparameters are *rank*=150 and *regParam*=0.05 which achieved the highest MAP=0.751 on the validation set.

Q5. Evaluation of latent factor model

We retrained the model with *rank*=150, *regParam*=0.05 on the Train and Validation sets, and evaluated the model performance on the Test set. The evaluation followed the same steps as the evaluation on the validation set and the popularity-based model.

MAP	NDCG at 5	NDCG at 100	Recall at 5	Recall at 100
0.751	0.829	0.891	0.406	0.956

We observe that the latent factor model outperforms the popularity-based model across all five evaluation metrics.

Contributions

Abhinav Krishnan - Built the popularity-based model and implemented the Ranking Evaluation metrics for Q4 and Q5.
Jason Moon - Implemented train-validation-test split, fine-tuned ALS hyperparameters using train and validation set, and evaluated the final ALS model on the test set.
Raj Raman - Found the top 100 similar “movie twins” and the correlation between these twins and the 100 random pairs (Q1 and Q2)