DS-GA 1004 – Big Data

Capstone Project

Group 23

Mason Lonoff

I. Introduction:

This project has two main components. The first portion consists of segmenting users with similar movie watching patterns. I utilized MinHashing (MinHash) and Locality Sensitivity Hashing (LSH) to identify pairs of users with overlapping interests. These pairs were then evaluated by comparing their rating correlations to the rating correlations of randomly selected pairs.

The second portion consists of creating a movie recommendation system. I implemented a baseline popularity model that recommends the top 100 most watched movies to each user. I also built two versions of a latent factor model using Spark's ALS. I tuned these models by cross validating the NDCG accuracy across the various hyperparameter values. I evaluated my models using Precision@100, Recall@100, NDCG@100, and MAP@100.

II. Part One – Customer Segmentation

The first major goal of this project was to identify users with similar movie watching habits. To achieve this at scale, I implemented a two-step approximate similarity algorithm using MinHash and Locality Sensitivity Hashing (LSH). This file can be found under minhash.py file in the repository.

Data Processing

To begin, I loaded in my ratings dataset and converted it to a parquet file. I then transformed the Data Frame so that every user had a list of every movie he/she watched. I then filtered out users that had rated 3 or fewer movies to improve reliability of my algorithm.

MinHash Signature Generation

To generate approximate Jaccard similarity scores between each user in a user pair, I applied a MinHash function. I experimented with different numbers of hash functions, and I ultimately settled on 128 hashes. This value allowed me to strike a balance between accuracy and speed which was a primary concern all project. My hash function took the form:

$h(x) = (a(x) + b) \bmod prime$,

where a and b are randomly generated integers and prime is a large prime number. Each hash signature was stored as a 128-dimensional vector. The vector was combined with the userId and the list of movies that were watched by each user.

LSH Banding

Each signature was divided into 32 bands of 4 rows each. If two users shared every hash value within a band, they were considered a collision and grouped together. LSH was used due to how it increases the chances of collisions for similar users and discourages collisions for dissimilar users.

Top 100 Pair Extractions and Validation

From these LSH buckets, I ranked the number of collisions (number of shared bands) in descending order and extracted the top 100. These results can be found in the top_100_outputs.txt file in the results folder on my project repository. The file shows the top 100 user pairs with their collision counts, the exact movie set that each user watched, and the Jaccard similarity.

To validate my results, I calculated the Pearson correlation of their movies ratings compared to a set of 100 randomly selected user pairs. The output file validate_pairs_output.txt shows that my clustered group has an average correlation of 0.06 while the randomly selected group has a correlation of 0.16. This indicates that the MinHash + LSH method does not capture how users actually enjoy each movie; it only captures watching behavior. People that watch the same movies don’t necessarily enjoy the same movies.

As a note, 6 rows were filtered out of my MinHash + LSH method when correlations were calculated. This was due to one (or both) of the movies not having any variance in their ratings. The Pearson correlation can’t be calculated with zero variance in one of the user’s movie ratings. These results can be found in the validate_pairs.txt file in the outputs folder.

III. Part Two - Movie Recommendation

To build a functional recommendation system, I approached the scenario in three dimensions. First, I preprocessed and then split the data. I then implemented a popularity-based model as a baseline, and finally, I developed two ALS based models based on either explicit or implicit ratings.

Data Preprocessing and Splitting

For preprocessing, I filtered out users who had less than five ratings and movies that had been rated less than five times. This was done to ensure data reliability.

For partitioning the data into training, validation, and test sets, I ordered each user's ratings in chronological order. I split each user's data 60/20/20 based on these timestamps to mimic reality where recommendations are made on past viewing behavior for each user. The resulting partitions were saved to parquet files to be reused efficiently in later phases. Upon analyzation, I found that there was zero overlap between any of the data in the three sets, so I was able to prevent data leakage. The code for this process can be found in split_ratings.py in my repository. The sanity checking output can be found in the validate_splits_output.txt file in the output folder.

Popularity Baseline

As a baseline, I used a simple popularity model that recommended the top 100 most popular movies in the training data to all users in the test set. Evaluation of these recommendations on the test set was done using Spark's *RankingMetrics.* The baseline model achieved 0.0215 Precision@100, 0.1839 Recall@100, 0.0866 NDCG@100, and 0.0230 MAP@100. The script for this file is baseline_model.py. The results of this model can be found in baseline_output.txt in the results folder.

Explicit ALS Model

I then used an explicit ALS model using Spark's ALS class with the user's ratings considered the user input. Two hyperparameters were tuned, *rank* and *regParam*, and the parameters were evaluated on the validation set using NDCG@100. I chose to use this as my benchmarking metric since NDCG captures correctness and the rank of the recommendation. Most users did not have a large number of ratings, so I wanted a metric that rewards earlier correct recommendations. The parameters that returned the best results were *rank = 30* and *regParam =* 0.05. My results were much worse than the baseline model. My results on the test set were 0.0004 Precision@100, 0.0065 Recall@100, 0.0019 NDCG@100, and 0.0002 MAP@100. This type of result indicates that raw user ratings may not be good enough to offer generalization in this situation. The code for this model can be found in als_model.py. The results of the tuning and accuracies can be found in als_results.txt in the results folder.

Implicit ALS Model

Due to the lackluster results above, I decided to reapproach this problem. I was going to approach this issue using implicit feedback. Now, each rating was treated as a binary preference (rating = 1 if exists) which was weighed by a confidence score. I introduced a new hyperparameter, *alpha,* which I also tuned along with *rank* and *regParam*. For this model, the ideal hyperparameter values were *rank = 25, regParam = 0.15,* and *alpha = 10* with a value of 0.1386 for NDCG@100. When I tested it on the model, I got 0.0245 for Precision@100, 0.2616 for Recall@100, 0.1023 for NDCG@100, and 0.0223 for MAP@100. These improvements validate that focusing on implicit behavior was beneficial for recommending movies. The code for this file is called implicit_als.py. The resulting output file can be found in implicit_als_results in the outputs folder.

Part IV: Conclusion

The project demonstrated the value of both approximate similarity algorithms and latent factor models for customer segmentation and movie recommendations. Through MinHash and LSH, I was able to segment users into distinct groups based on their viewing behavior. While clustering worked well, it was clear that shared viewing behavior does not imply shared preferences. When it comes to movie recommendations, the baseline score was generally a decent way to recommend movies. The implicit model proved to be the most effective at recommendations though. There is potential in the latent model to be improved upon. The project continuously highlighted the tradeoffs between complexity and efficiency and was a valuable learning experience.