Preprocessing, Popularity-Based Recommender, and Evaluation Metrics for Music Recommendation System

ABSTRACT

In this report, we present the methods used to aggregate the interaction count data, create a popularity-based recommender system, and evaluate the performance of the system. Our preprocessing steps ensured that the data was clean and ready for analysis, while our evaluation metrics allowed us to assess the quality of the recommendations generated by the system. The performance of the recommender system increased with higher values of beta. As part of future work, we will explore other recommendation algorithms and techniques, as well as investigate methods to incorporate user and song metadata to enhance the quality of recommendations.

CCS CONCEPTS

• Information systems → Recommender systems; • Computing methodologies → Machine learning algorithms.

KEYWORDS

Music Recommendation System, Preprocessing, Popularity-Based Recommender, Evaluation Metrics, Interaction Count Data

1. PREPROCESSING:

We implemented the following steps for preprocessing of the data:

- Read the tracks, interactions, and users parquet files from HDFS and converting them to dataframes.
- Joined the interactions and tracks dataframes on the 'recording_msid' column.
- Joined the resulting dataframe from the previous step with the users dataframe on the 'user_id' column.
- Used the Pandas library for data manipulation to deal with null values in the 'recording_mbid' column which were directly replaced with the corresponding 'recording_msid' values.
- Dropped columns which were not going to be used in the subsequent steps

2 IMPLEMENTATION:

2.1 Popularity - Based Recommender System

◆ Data Splitting

To create the training and validation sets, we performed the following steps:

- Sorted the data by timestamp for each user.
- Calculated the split point for each user based on the specified split ratio (0.8).
- Split the data into training and validation sets for each user respectively.
- Saved the training and validation data to disk as Parquet files.

◆ Aggregating Interaction Count Data

We used the following methods to aggregate the interaction count data:

- Loaded the training and validation data from the Parquet files.
- Grouped by ‘recording_mbid’ and ‘user_id’ to obtain the total count of interactions for each user and recording.
- Saved the aggregated count data to disk as Parquet files.

◆ Methodology

We created a popularity-based recommender system by implementing the following steps:

- Read the aggregated count data for the training set.
- Computed the popularity score for each song, by taking the total number of interactions for each song divided by the unique number of users that have listened to that song plus some damping factor (By incorporating the beta hyperparameter, the model can control the influence of the number of users on recommendation scores)
- Selected the top 100 most popular songs based on the computed scores.
- Saved the top 100 songs as a Parquet file.

◆ Evaluation

We used three evaluation metrics to assess the performance of our recommender system :

1. Mean Average Precision (MAP): MAP measures the average precision of recommended songs across all users. It considers the order of the recommendations and rewards systems that rank relevant items higher.
2. Precision at k (P@k): P@k measures the proportion of relevant items across the top-k recommended songs. It does not consider the order of the recommendations.
3. Normalized Discounted Cumulative Gain at k (NDCG@k): NDCG@k measures the ranking quality of the top-k recommended songs, accounting for the

relevance of the items and penalizing systems that rank relevant items lower.

We trained the popularity based recommendation model on the training data using 4 different values of the hyperparameter beta (100, 1000, 10,000 and 100,000) and evaluated it on the validation and test data on 4 different values of k (10,20,50,100).

Our results show that the performance of the recommender system increased with higher values of the hyperparameter beta, while the P@k and NDCG@k metric increased with increasing values of k. Hence, The best configuration found for our popularity-based recommender system was with beta=100,000. The MAP score remained same over every k achieving a value of 0.00010 on the validation data and 1.7767e-05 on the test data. Precision at k was highest at k=100 with a value of 0.02334 on the validation data and a value of 0.00812 on the test data. Normalized Discounted Cumulative Gain at k was highest at k=100 with a value of 0.00142 on the validation data and a value of 0.00038 on the test data.

2.2 ALS Recommendation Model

◆ Data Preprocessing

- **Compressed Interactions Data:**

To improve our analysis of the interactions dataframe, we applied the log1p function to the count column. This helped compress the values, making the data more manageable and preparing it for further analysis.

By compressing the data, we achieved multiple benefits.

- Firstly, the compression enabled us to normalize the count values and therefore facilitated easier analysis and interpretation. This is because normalizing the values eliminated the potential for bias in our analysis, ensuring that we could compare values accurately.
- Secondly, this compression helped address the skewness that is often present in count data, ensuring that extreme values did not overly influence our recommendations. This is important because extreme values can skew the results of our analysis, making it difficult to draw accurate conclusions.
- Additionally, converting counts to ratings helped capture non-linear relationships between user preferences and item popularity. This transformation improved the performance of the ALS algorithm, which relies on rating-based data. This means that our recommendations were more accurate and personalized for our users.
- Ultimately, by compressing counts to ratings, we were able to enhance the quality of our recommendations, leveraging additional information and ensuring a more personalized experience.

- **Indexing Recording MBIDs:**

To ensure compatibility with the ALS model, we had to develop a new process for preparing the songs' identifiers, which we then converted from string (Alpha-numeric) data to numerical indices. The ALS algorithm relies on numeric indices to identify items and users in the recommendation process, so this was a crucial step.

To assign unique numerical indices to each song, we used a dense rank approach. This involved analyzing the data and assigning a rank to each song based on its popularity and other key factors. By using a dense rank approach, we were able to maintain the order and density of the original data without creating any gaps or missing indices.

This indexing strategy was crucial to our work with the ALS model. By using numerical indices, we were able to seamlessly integrate our data with the ALS algorithm, which in turn allowed us to provide more accurate and personalized music recommendations to users. The processed numerical indices provided a more efficient way of analyzing and categorizing the music in our database, which ultimately led to better recommendations for our users.

◆ Model Training and Evaluation:

We conducted hyperparameter tuning on the ALS model to optimize its performance. This process involved exploring different combinations of rank, regularization, and alpha values in order to improve the model's accuracy and recommendation quality. Specifically, we tested a range of values for each hyperparameter, and then evaluated the resulting models based on their performance metrics.

After training the ALS model on the indexed training dataset, we generated personalized recommendations for all users in the validations dataset based on the model's predictions. This involved using the model's predictions to identify the most relevant items for each user, based on their preferences and past behavior. To evaluate the quality of our recommendations, we utilized the Mean Average Precision (MAP) metric. This metric measures the model's ability to rank relevant items higher, and is calculated by comparing the top 100 recommended items for each user with the top 100 songs in the test set.

Rank	**Alpha**	**Regularization**	**MAP**
10	60	0.1	0.189186
10	40	0.25	0.189733
10	40	0.5	0.190037
10	10	0.01	0.262888
10	10	1	0.262887
10	40	0.5	0.262998

20	40	0.5	0.263065
15	10	0.5	0.263086
10	5	0.5	0.263194

Overall, our hyperparameter tuning process yielded the best results with a rank of 10, alpha of 5, and regularization of 0.5. This configuration produced a MAP score of 0.263194 for the validation dataset. Once we had identified the optimal hyperparameters for the ALS model, we trained it on the indexed dataset again and generated personalized recommendations for all users in the test set achieving a MAP score of 0.169263.

3. EXTENSIONS:

3.1 LensKit ALS Recommendation Model

For the next part of our project, we decided to try out the LensKit ALS model as an alternative to the previously used ALS model. We were excited to explore the potential benefits of using this new model and to compare its performance to that of the Spark ALS model.

To begin, we needed to prepare the data in the required format for the LensKit ALS model. We wanted to ensure that we had a comprehensive understanding of the model's capabilities, so we decided to randomly select different percentages of the training data. We started by selecting only 10% of the data and then increased the percentage in increments of 10% until we had used the entire dataset. By doing this, we were able to explore how the performance of the model varied with different amounts of training data.

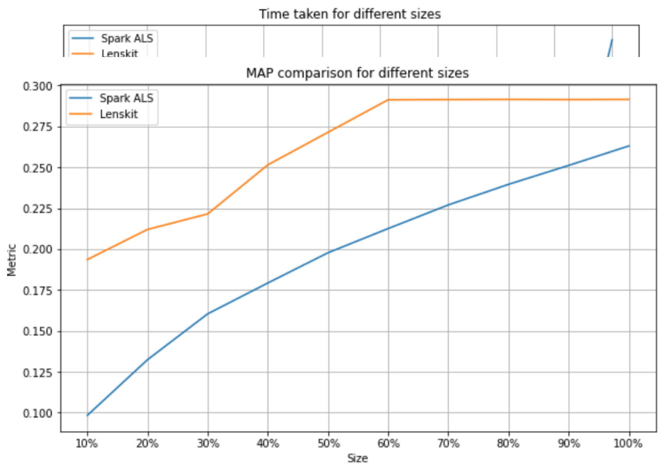
Next, we needed to ensure that the data was formatted correctly for the LensKit ALS model. To do this, we renamed the columns in the data to 'user', 'item', and 'rating' in order to match the format expected by the LensKit ALS model.

With the data formatted correctly, we were ready to fit the LensKit ALS model and obtain predictions for the test set. We were interested in comparing its performance to that of the Spark ALS model. We performed the evaluation process, which included generating personalized recommendations and calculating the Mean Average Precision (MAP) score, in a similar manner as with the Spark ALS model.

To evaluate the performance of the LensKit ALS model, we created plots that displayed the comparison of execution time and MAP scores for different sizes of the dataset. We were curious to see how the model's performance would vary with different sizes of the dataset.

The results showed that as the size of the dataset increased, both the execution time and the MAP scores varied for both models.

After evaluating the Spark ALS and LensKit ALS models using the test data, we obtained the following results. For the Spark ALS model, the algorithm took approximately 184.40 seconds to implement, and achieved an MAP score of 0.169263. On the other hand, the LensKit ALS model required a longer implementation time of around 1084.97 seconds, with an MAP score of 0.189873.



These results provide insights into the performance of the two models in terms of their implementation time and their ability to generate accurate and relevant recommendations.

3.2 Recency-Weighted Model

- **Implementation**

With our current implementation of ALS, the time of interactions is not taken into account by the recommender system. More specifically, the interactions for one user are all given the same weight when counted, regardless of whether that interaction occurred yesterday or thirty years ago. We hypothesized that the recency of an interaction *does* matter, because people's likes and interests change over time. Thus we explored altering the weight of an interaction between a user and recording on our recommender system, providing more weight to more recent interactions. We did this in the following manner:

Rather than counting the number of interactions between a user and recording, instead a weight of recency was calculated for an interaction based on the user's previous interactions. For each user, interactions were ordered by timestamp descending and set an index based on this order. For example: if a user had *n* interactions the indices would be created such that their first (most recent) interaction was given an index of 0, their second-most recent interaction was given an index of 1, and so on until their last (least recent) interaction was given an index of n-1. (Note that "index" here refers to a new created column called "index" with calculated values, rather than the relational DB definition of indexing.) Then, a *recency weight* was set based on this index term for each user-recording interaction. Specifically, the weight was set to be $e^{(1-index)}$. Thus, the most recent interaction would be given a recency weight of *e*, the second-most recent a weight of *1*, the third-most recent a weight of $\frac{1}{e}$, the fourth-most recent a weight of $\frac{1}{e^2}$, so on & so forth.

◆ Evaluation

We trained an ALS recommender model on the provided "small" dataset (due to time and resource constraints) using recency to weight interactions. In the future we look forward to implementing our extension against the full set of data. The code to order records per user by decreasing order of timestamp, add an index per user depending on this order, and calculate an index-based weight for each user-recording interaction can be found in the files that contain titles ending in "*recency*" on GitHub. The following table compares the results of our recency-weighted model on the small dataset against the normal frequency ALS model on the small dataset, using the Mean Average Precision:

Rank	Alpha	Regularization	MAP **RECENCY MODEL (extension)**	MAP **COUNT MODEL (old)**
10	60	0.1	0.184223	0.189186
10	40	0.1	0.185101	0.189733
10	40	0.5	0.186984	0.190037

◆ Discussion & Limitations

Incorporating the notion of recency into our recommender system to penalize older interactions did not seem to increase certain performance metrics of the model. Specifically, Mean Average Precision (MAP) decreased slightly compared to when the model used counts between users and recordings to capture each user's preferences. This essentially means that capturing interaction recency yielded in recommendations of slightly less relevant songs than when count of user-recording pairings were being used. In other words, interaction frequency seems to be a good predictor of liked recordings compared to interaction recency.

There are certain limitations to consider when discussing this extension. First of all, the only evaluation metric used to measure the performance of our models was Mean Average Precision (MAP), and we did not use Precision @ k or Normalized Discounted Cumulative Gain @ k. It would be interesting to evaluate with NDCG@k in the future as this metric would translate to an evaluation of ranking quality. We hypothesize that it would be higher for a recommender that considers recency. An additional limitation has to do with the mathematical methods used to capture recency. Notice that by the seventh-most recent practice, the weight has reduced to $\frac{1}{e^5}$ = 0.0067. By the twelfth-most recent practice, the weight has reduced to $\frac{1}{e^{10}}$ = 0.000045. The only interactions contributing significantly to the recommendation are likely the four or five most recent interactions. We believe this is not truly representative of how recency provides insights on current preference, as this method essentially loses any interactions that pass a level of recency. To combat this, at least two solutions could be implemented. First, we could model recency using a much more gradual penalization function: for example, $weight = e^{(1-(0.01 * index))}$. Secondly, we could set a minimum threshold on weight (for ex.: 10^{-6}) to ensure we do not lose any interactions data, and that interactions which are not recent are simply given the same weight rather than having the weight converge to 0 for these interactions. Our group looks forward to exploring how these changes affect the evaluation of our recency model in the future.