DS-GA 1004 BIG DATA

GROUP 42

https://github.com/nyu-big-data/final-project-group-42/

Kritik Seth
Center for Data Science
New York University
New York, NY, USA
kls8193@nyu.edu

Saahil Jain
Center for Data Science
New York University
New York, NY, USA
sbj7913@nyu.edu

1. Abstract

The goal of this project is to build and evaluate a recommender system on the ListenBrainz [1] dataset. To start off, we develop a baseline popularity model based on a time decay popularity metric we define ourselves. We then build and evaluate a recommender system using PySpark's ALS module to create more personalized recommendations. Further to extend the project, a model is built and evaluated using LightFM [4], and its performance is compared to the ALS model. The project aims to leverage the power of recommender systems to predict user preferences and enhance the recommendation experience for various music services.

2. Data Preprocessing

Data Preprocessing involved the following steps:

i. Joined interactions and tracks dataset on “recording_msid”
ii. Imputed the null values in “recording_mbid” with corresponding value with “recording_msid”
iii. Created a new dataframe with columns- “user_id”, “recording_mbid”, “timestamp”.

3. Train Validation Split

We used a Stratified split for each user to divide our data into train and validation such that for each user 70% of their interactions were in train and 30% in validation. We further tried two methods to decide which interactions would belong to which set. The two methods are as follows:

Method 1: Stratified Split based on Timestamp

i. A new column named "row_number" was created, which assigned a unique number starting from 0 to each interaction of a user, ordered by timestamp from oldest to most recent.
ii. A new column named "max_row_number" was created for each user, containing the value of the last row in the "row_number" column.
iii. A new column named "split_index" was created by multiplying "max_row_number" with a fraction indicating the ratio of "train_data" to "all_data".
iv. The train dataset was created by selecting all rows for every "user_id" where "row_number" was less than "split_index".
v. The validation dataset was created by selecting all rows for every "user_id" where "row_number" was greater than or equal to "split_index".

Method 2: Stratified Random Split

i. The new column named "row_number" was created, which assigned unique numbers to each interaction randomly (starting from 0) - instead of being ordered by timestamp - for each user.
ii. [Repeat steps 2-5]

While we explored two approaches for creating the train-validation split. Initially, we observed better performance on randomly split data. However, considering the deployment scenario where the model needs to predict based on the user's actual past history, we opted for a Stratified Split based on Timestamp. This approach involved taking the first 70% of each user's data as the training set and the remaining data as the validation set. This stratified split based on timestamp ensures that the model is trained on a user's earlier interactions and evaluated on their more recent interactions, closely resembling real-world prediction scenarios.

4. Baseline

We tried multiple popularity baseline models such as average play per user, total plays per user, through total unique interactions, and time-based but the best popularity baseline in terms of highest MAP was time-based popularity baseline model.

Dataset was grouped by “recording_mbid” and a time decay function was calculated and applied as follows.

$$w(t) = e^{(\frac{-\lambda \cdot t}{86400})}$$

here t is an estimate of the amount of time music has been available, λ is the decay parameter and 86400 is the number of seconds in a day. This estimate of t was calculated by grouping on “recording_mbid” and subtracting the lowest timestamp for each music from the highest. We multiplied this time decay function with the

number of unique users who interacted with the song to calculate our “popularity” parameter. The dataset was then sorted on “popularity” (highest to lowest) and top 100 rows from “recording_mbid” were selected to be our recommendations.

Our rationale behind this approach is that if a song released yesterday receives the same level of interactions as a song that has been available for a year, it is likely to be more popular in recent times and therefore a better candidate for recommendations.

4.1. Hyperparameter Tuning

We adopted a time-based approach and conducted hyperparameter tuning. We explored a range of fourteen values for the hyperparameter (λ), ranging from 1e-05 to 0.5. For each value, we calculated the Precision at K (10, 50, 100) on both the training and validation datasets. The value that yielded the highest Precision at K on the validation dataset was selected as the best value, which turned out to be 1e-05. It is important to note that this observation was made on an ordered split dataset.

Dataset	λ	Precision At			Mean Average Precision
		10	50	100	
Train	1e-05	.0926	.0837	.0756	.00053
Validation		.0502	.0447	.0399	5.041e-05
Train	5e-05	.0926	.0837	.0756	.00053
Validation		.0502	.0447	.0399	5.060e-05
Train	1e-04	.0926	.0837	.0756	.00053
Validation		.0502	.0447	.0398	5.134e-05
Train	5e-04	.0910	.0816	.0744	.00050
Validation		.0490	.0440	.0396	6.891e-05
Train	1e-03	.0743	.0716	.0651	.00028
Validation		.0432	.0415	.0377	9.579e-05
Train	5e-03	.0108	.0089	.0082	.00012
Validation		.0266	.0216	.0186	.00013
Train	1e-02	.0052	.0054	.0047	6.533e-05
Validation		.0114	.0125	.0112	.00010
Train	5e-02	.0017	.0014	.0013	2.684e-05
Validation		.0011	.0008	.0008	6.119e-05
Train	1e-01	.0012	.0010	.0008	9.399e-05
Validation		.0008	.0005	.0003	2.430e-05

4.2. Results

In the optimization process, we noticed that the precision values were generally similar for most values. However, when examining the precision values in lower decimal places, we found that the highest precision was observed at a value of 5e-05. Finally training the baseline model with this λ on the entire train data produces a model with the following performance on Test:

λ	k	Precision at k	
		Train	Test
5e-05	10	.1057	.0061
	50	.0959	.0077
	100	.0872	.0088

5. Latent Factor Model

The Alternating Least Squares (ALS) algorithm is a collaborative filtering technique used to make recommendations. It works by decomposing the user-item interaction matrix into latent factors for users and items. The algorithm iteratively alternates between optimizing the user factors while holding the item factors fixed, and vice versa. This process continues until convergence, allowing the algorithm to predict user ratings or preferences for items they haven't interacted with. This enables personalized recommendations.

Since our data only contained implicit feedback in the form of interactions, we created our own metric for rating as count of interactions. So for each user, the rating associated with a specific song is the number of times the user interacted with it. Finally creating our user-item-rating matrix on which we could use the ALS algorithm to build a personalized recommender system.

5.1. Hyperparameter Tuning

To tune the model to the best possible performance, we tried a variety of combinations of rank and regParam hyperparameters and trained and evaluated them on our train-validation split. We did face some issues training and evaluating the model with high rank on the cluster, and hence had to limit our optimization search to a maximum rank of 50, This is also the reason we don't have results for all combinations of rank and regParam. WOur results were as follows:

Rank	Reg Param	Precision At	Mean Average Precision	Time (s)
		10		
10	0.01	.384	.226	294
	0.1	.383	.237	262
	0.2	.383	.226	281
20	0.1	.191	.119	394
	0.2	.400	.235	523
35	0.1	.414	.235	885
	0.2	.436	.241	1907
50	0.01	.420	.237	921
	0.1	.422	.237	2387

5.2. Results

Although we observed the best Precision at rank 50, we faced storage issues on Dataproc forcing us to choose the next optimal rank of 35, with a regularization parameter of 0.2. Finally, we took these parameters, and trained a new model on the entire test data and observed the following performance on the test data:

Training Time	k	Evaluation Time (s)		Precision at k	
		Train	Test	Train	Test
3876	10	0.1	0.1	.466	.336

6. LightFM

To compare the efficiency and accuracy of the LightFM and ALS algorithms, we implemented LightFM, a Python recommendation algorithm, on a single machine using the official documentation of LightFM version 1.16. To ensure a single-machine implementation, we ran the program in a Singularity container on the NYU Greene cluster. We installed the required packages, including LightFM, Pandas, Pyarrow, and Fastparquet, within the Singularity container.

For each run, we submitted a slurm job requesting 1 node and 1 CPU per task. To ensure a single-machine execution, we set the num_thread hyperparameter to 1 in LightFM.

By following these steps, we investigated the efficiency and accuracy differences between LightFM and ALS algorithms.

6.1. Hyperparameter Tuning

In contrast to Spark ALS's rank and regParam hyperparameters, we used LightFM's no_components (dimensionality of feature latent embeddings) and alpha as hyperparameters. We performed experiments on small and large datasets, tuning the hyperparameters to compare with Spark ALS later.

num components	α	Precision At			Time (s)
		20	50	100	
20	0.01	.585	.335	.173	934
	0.1	.616	.354	.193	934
	0.5	.623	.348	.194	934
50	0.01	.593	.319	.175	2307
	0.1	.618	.354	.193	2307
	0.5	460	.234	.147	2307
100	0.01	.461	.223	.116	3619
	0.1	.456	.219	.115	3619
	0.5	.456	.222	.116	3619

6.2. Results

Training Time	k	Precision at k	
		Train	Test
3876	10	.762	.602
	50	.426	.363
	100	.330	.220

7. Conclusion

After analyzing the evaluation time and metrics of our baseline model, ALS Latent Factor Model, and LightFM using a single machine implementation, we determined that LightFM outperformed the other models in terms of Precision at 10. It achieved a Precision at 10 of 0.602 which was significantly higher compared to the ALS Latent Factor Model and Baseline Popularity model. On the other hand, this increase in performance came at the cost of training time. Training time of the LightFM model was more than twice as that of the ALS Latent factor model, and took about 2287 seconds on average which is 2.5 times the average time of 872 seconds taken by the ALS Model.

8. Contributions

Kritik Seth - Data Cleaning, Baseline and LightFM

Saahil Jain - Splitting, Baseline and ALS

9. References

i. “ListenBrainz Data.” n.d. ListenBrainz. Accessed May 16, 2023. https://listenbrainz.org/
ii. “Overview - Spark 3.4.0 Documentation.” n.d. Apache Spark. Accessed May 16, 2023. https://spark.apache.org/docs/latest
iii. McFee, Brian, and Pascal Wallisch. n.d. “NYU Big Data Course.”
iv. “LightFM.” n.d. LightFM 1.16 documentation. Accessed May 16, 2023. https://making.lyst.com/lightfm/docs/home.html