Collaborative filter recommendation system on ListenBrainz

Dian Jin
Center for Data Science
New York University
dj928@nyu.edu

Xinxiaomeng Liu
Center for Data Science
New York University
xl4701@nyu.edu

Abstract

In this final project, we built and evaluated a collaborative-filter based recommender system using the ListenBrainz dataset. This data consists of implicit feedback from music listening behavior, spanning several thousand users and tens of millions of songs. Each observation consists of a single interaction between a user and a song.

We preprocessed the data by first joining the DataFrames, then performing time-based user-wise splitting to partition the training data into `train` (75% of total) and `val`(25% of total). We also simplified the datasets to retain only the necessary information, by mapping recording_msid to integers and calculating the interactions between each (user_id, recording_msid) pair.

We applied two models: a popularity baseline model and an ALS model, to predict the top 100 recording_msid. Our baseline model predicts the top 100 most listened to recording_msid based on the total counts of every recording_msid, which is the number of times each recording_msid has been listened to in total. We used two different evaluation metrics: Mean Average Precision (MAP) and Normalized Discounted Cumulative Gain (NDCG), to evaluate the models' performance. To improve performance of our ALS model, we used Grid Search to do hyperparameter tuning. We tuned hyperparameters including the rank (dimension) of the latent factors, the regularization parameter and the implicit feedback parameter (alpha).

Then we compared the ALS model and a single-machine-implementation: LightFM. The set of parameters for constructing the LightFM model was also chosen through grid search where we distinguished the parameters that were most relevant to the model and those that did not bring significant impact to the performance metric, which is the average precision at rank = 100. We concluded that the LightFM model performs generally better than the latent factor model and discussed the possible reasons for this phenomenon.

Data Preprocessing and Partitioning

In order to proceed with subsequent steps of building and testing models, we needed to preprocess the original data into an appropriate format and extract a training set, a validation set and a test set separately.

We first loaded the small Parquet files storing data for interactions, users, and tracks as Pyspark dataframes. Interactions and users DataFrames are joined on `user_id`. The resulting DataFrame is joined by tracks on `recording_msid`, so that for each row of data we would have all the information that was available for each user-item interaction. Parquet files on testing data were joined in the same way.

We then performed time-based user-wise splitting to partition the training data into `train` and `val`. A new column named `timestamp_ms` is computed as a Unix timestamp in milliseconds. We used window functions and `percentile_approx` to compute the 75% quantile of timestamp for each user. Per user interactions with older timestamps that constitutes 75% of total train data into the training set. All of the remaining 25% is stored as the validation set. The logic is that if the training set contains data from the past while the validation set contains more recent data, the popularity based system can learn from history and predict future trends more accurately. Also, since historical and more recent data are divided per user, we ensure that both the training and validation sets have representations of each user's behavior to avoid user cold start problems.

The next step was to simplify the datasets to retain only the necessary information in their most simple form. We created a Spark DataFrame that was composed of all the data from training, validation and testing, and used a Window function along with `dense_rank` to substitute each 'recording_msid' with a distinct integer. Then, the separated training, validation and testing datasets had their 'recording_msid' columns mapped with these newly assigned integers. The original recording id columns were dropped. In this way, we avoided processing long string values in our analysis, which improved storage efficiency, memory usage, and overall enhanced performance of operations. Moreover, data privacy was better protected as we put a mask over the recording details. We noted that while this replacement has many advantages, it was crucial to maintain the integrity and uniqueness of the data. With this purpose in mind we kept a lookup table associating the integer values with the original string values to ensure proper interpretation and analysis of the data.

The last step before writing the files to output, was to impose an aggregate function to count the number of interactions between each existing user-item pair. This would act as an implicit factor to provide insights into user preferences and item popularity. This aggregation operation produced a high-level summary of the data, significantly reduced the storage requirement, and enabled more efficient data retrieval and faster computation. Below is a snippet of the resulting data format.

user	item	count
291	1071552	17
11443	1074190	4
21490	1086280	1

Table 1 Data sample

Eventually, the dataframes were written to HPC as CSV files. We previewed the data by showing the first five rows to check if everything was in place.

Popularity Baseline Model

We first built a popularity baseline model, which can provide a baseline score for the ALS Model. Our baseline model predicts the top 100 most listened to recording_msid simply based on the total counts of every recording_msid, that is, how many times each recording_msid has been listened to in total. We applied the model to the full training dataset, predicted the top 100 most listened to recording_msid and converted the 100 recording_msid to an array of double values.

For the validation set, we generated the top 100 recording_msid for each user_id by first defining the window function to partition by user_id and sort by count (how many times each recording_msid has been listened to by each user_id) in descending order and assigning row_number to each recording_msid for each user based on the count, then filtering for the top 100 recording_msid for each user and converting each user's top 100 recording_msid to an array of double values. We used meanAP@100 and NDCG@100 with *pyspark.ml.evaluation.RankingEvaluator* to evaluate the model performance.

For the test set, we did the same operations as we did for the validation set to generate the top 100 recording_msid for each user_id. Then we converted each user's top 100 recording_msid to an array of double values and used meanAP@100 and NDCG@100 to evaluate the model performance.

	MAP	NDCG
Validation	5.3185226E-05	0.001058803637
Test	0.000141683954	0.002233094151

Table 2 MAP@100 and NDCG@100 of the popularity baseline model

At the beginning, we also tried another baseline model, where we predicted the top 100 most listened to recording_msid based on the average number of listens per user, that is, dividing the total number of listens by the number of distinct users plus a damping value. We tried damping values in the range from 10 to 10000. But the performance of the model was never good no matter how we adjusted the damping value.Therefore, we chose the simple counts based baseline model we described above, which predicts the top 100 most listened to recording_msid simply based on the total counts of every recording_msid. The logic behind the average-number-of-listens-per-user based model is that we treat the counts of interactions between a user and a recording as rating, and we choose the highly-rated recording_msid, that is, the recording_msid that has been listened to most by the user who have listened to it. However, the total number of people who have listened to it also shows the popularity of a song. Therefore, it's reasonable to choose the counts based model we described above, which takes every interaction between user and recording into account.

Alternative Least Squares Model

We built the ALS model using spark's *pyspark.ml.recommendation.ALS()*. To maximize the performance of the ALS model, we tuned the hyperparameters: the rank (dimension) of the latent factors, the regularization parameter and the implicit feedback parameter (alpha),. We used Grid Search to find the best parameters. The range of values to search for $(rank,\ regParam,\ alpha)$ is as follows:

params_grid = {
'rank': [50, 100, 150, 200],
'regParam': [0.01, 0.1, 1.0],
'alpha': [0.1, 1.0, 5.0]
}

We evaluate the model's performance on the validation set using meanAP@100 as the evaluation criteria. We extracted 1/1000 rows of the original datasets to train the ALS model because the runtime is extremely long if we use the full dataset. In order to search over more hyperparameter combinations, we made this compromise. Although reducing the dataset has a negative impact on the performance of the model, the reduced dataset is still large enough to train the model, so this approach works.

We found that alpha has limited influence on the model's performance. To save space, we only display the performance of the model with alpha = 1.0.

rank	reg	alpha	val_MAP
50	0.01	1.0	5.1609E-06
50	0.1	1.0	1.1513E-05
50	1.0	1.0	1.1513E-05
100	0.01	1.0	3.4187E-05
100	0.1	1.0	3.4758E-05
100	1.0	1.0	3.3018E-05
150	0.01	1.0	1.5056E-05
150	0.1	1.0	1.4883E-05
150	1.0	1.0	1.6043E-05
200	0.01	1.0	1.5790E-05
200	0.1	1.0	1.4340E-05
200	1.0	1.0	1.4379E-05

Table 3 MAP@100 of the ALS model with different hyperparameter combinations

The best combination of hyperparameters we found is $(rank,\ regParam,\ alpha)\ =\ (100,\ 0.1,\ 1.0)$. From the outcomes we can see that rank has a relatively large influence on the model's performance. To explore further, we searched over:

params_grid = {
'rank': [50, 60, 70, 80, 90, 100, 110, 120, 130, 140, 150],
'regParam': [0.1],
'alpha': [1.0]
}

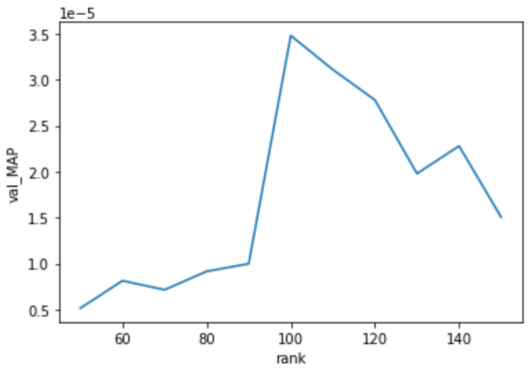

Figure 1 MAP@100 of the ALS model with different ranks

From Figure 1 we can see that MAP@100 tends to increase as rank increases from 50 to 100 and decreases as rank increases from 100 to 150. Therefore, $(rank, regParam, alpha) = (100, 0.1, 1.0)$ can be used as the optimized combination of hyperparameters. The MAP@100 and NDCG@100 of the ALS model with the optimized hyperparameters on the full dataset (without being reduced) is as follows:

	MAP	NDCG
Validation	0.000264723492	0.007674394723
Test	0.000424734245	0.009356997644

Table 4 MAP@100 and NDCG@100 of the ALS model

By comparing the MAP@100 and NDCG@100 of the ALS model and that of the popularity baseline model, we can see that the ALS model does perform better in terms of both evaluation criteria. This outcome is not surprising, as ALS often outperforms them due to its ability to capture personalized preferences, handle sparsity, and model complex relationships between users and items. However,our baseline model has its own merits. It's much simpler than the ALS model. And the runtime and computation resources needed for the popularity baseline model is much less than that needed for the ALS model.

Single Machine Implementation - LightFM

We proceeded with building a single machine implementation of the recommendation system using LightFM[1], which is a recommendation system algorithm that uses a combination of collaborative filtering and matrix factorization to make personalized recommendations. One of the key characteristics of LightFM is that the absence of interactions between a certain user-item pair is treated as meaningful information, instead of simply missing data. LightFM treats such absence as a deliberate choice to capture both explicit and implicit user preferences, as users might choose not to interact with certain items due to various reasons, such as lack of interest or unawareness.

We utilized the same training, validation and test datasets as the previous models. There were three columns to work with: user id, recording id and the number between their pairwise interactions. For validation and testing purposes, we extracted 1/100 rows of the original datasets at random to save time and computational resources. Using the full validation and test sets was too time consuming and resource-intensive in our single machine model, making the entire project impractical. Working with a subset of the data, we can get a reasonable estimate of the model performance while reducing computation burden.

First of all, we built sparse matrix representations out of the original datasets, where the rows correspond to each user, the columns correspond to each recording, and the values represent the interaction strength, which is the number of interactions in this project. This sparse format is used to store the large-scale recommendation data, where the majority of interactions are typically empty. Then, we performed grid search to find the best set of hyper-parameters using training and validation data. The specific parameters at work were `no_components` (the dimensionality of the feature latent embeddings), `user_alpha` (L2 penalty on user features), `item_alpha` L2 penalty on item features), and `loss` (type of loss function). There are more available parameters that can be adjusted but we only involved the mentioned parameters because we believed they tend to have a significant impact on LightFM performance and we had to take into account the practicality and complexity of tuning multiple parameters. We also tried to adjust the specification for `epoch`, hoping more training rounds would improve model performance. We found that the precision metric was not significantly improved while computation time experienced a surge. Therefore, we discarded the plan to include epoch in the grid search.

We recorded the precision at k=100 for each set, as well as the time cost to fit and evaluate the model as a measure of model efficiency. According to the LightFM documentation, there are a few choices for evaluating the model such as `auc_score` and `recall_at_k`. In our project, we chose to use `precision_at_k` and set k to 100 so that we could conveniently compare the performance of the LightFM model against the latent factor model in Spark. The `precision_at_k` computes the fraction of known positives in the first k position of the ranked list of results. We need an extra processing step to transform it into average precision for comparison with ALS. Below is a table of the selected result.

loss	user_alpha	item_alpha	Time (fit)	Time (evaluate)	Time (total)	AP@100
warp	1	0	10761	1588	12349	4.04095E-04
warp	0.1	0	1310	1601	2912	5.44218E-04
warp	0.01	0	170	2081	2251	4.01075E-04
warp	0.1	1	15354	1651	17005	6.75675E-05
warp	0.1	0.1	15282	1650	16932	1.72109E-04
warp	0.1	0.01	1613	1675	3288	1.29714E-04
bpr	0.1	0	1570	1668	3238	1.03288E-04

Table 5 Runtime and precision@100 for LightFM model

From the snippet of the result above, we can make a few observations. Time-wise, the difference between the sets of parameters had a relatively smaller impact on evaluation runtime than the time to fit the models. In particular, increasing the regularization parameters to a factor of 10 for user features and item features had a huge impact on training time. When the `item_alpha` was set to 0, we observed higher precision than when we specified values for it such as 1, 0.1 and 0.01. This suggests that the regularization on the item features might not be necessary or beneficial for our dataset and the absence of item feature regularization allows the model to focus more on the user-item interactions, potentially improving the precision of recommendations, which makes sense in our case because the original data did not provide detailed information on each music recordings. When `user_alpha` was set to 0.1, the precision was higher than when it was 1 or 0.01, which suggests that a user feature regularization factor around 0.1 strikes a balance between overfitting and enhancing the precision of the recommendations. Moreover, the precision was higher for `warp` loss than for `bpr` loss when other parameters were held the same. The warp loss function (Weighted Approximate-Rank Pairwise) is designed to optimize precision-oriented ranking measures that trains the LightFM model to optimize the ranking of positive interactions, which is the user-item pairs with observed interactions, compared to randomly sampled negative interactions. Naturally it was a better choice when precision was the most crucial evaluation metric in our project.

Final Result: according to the parameter tuning above, we chose this combination of parameters for the LightFM model:

- loss = ‘warp’
- user_alpha = 0.1
- item_alpha = 0.0
- epoch = 1

The performance on the testing set using these parameters is the following:

Time (fit)	Time (evaluate)	Time (total)	AP@100
1059	1985	3044	5.2169136E-04

Table 6 Runtime and precision@100 for LightFM test set

The resulting LightFM model was reasonably well trained in terms of the precision metric and runtime. We believe that LightFM could have generated better predictions since it has the ability to incorporate features beyond the user-item interactions, such as user demographics and content-based data, which was not available in our dataset. Comparing the test result of the Spark ALS model, we observed that average precision@100 was improved using the LightFM model. Possible reasons include that LightFM takes a hybrid approach that combines collaborative filtering and matrix factorization with item and user features, which allows it to leverage multiple signals for recommendations and to capture more nuanced user-item interactions. Also, LightFM provides a relatively larger set of parameters for tuning such as embedding dimensions and regularization terms, allowing customization of the model to better fit the experiment setting.

Github Link

https://github.com/nyu-big-data/final-project-group-80

Contribution

Dian Jin:

- Data preprocessing and partitioning
- Single machine implementation - LightFM

Xinxiaomeng Liu:

- Popularity Baseline Model
- Alternative Least Squares Model

Reference

[1] M. Maciej Kula. LightFM: A Python Library for Hybrid Recommender Systems. In: Proceedings of the 10th ACM Conference on Recommender Systems (RecSys), 2016. https://making.lyst.com/lightfm/