Music Recommender System

1 INTRODUCTION

In the `experiment` folder, we have three files:

- `als.py`: the base code for ALS that will be introduced in section 3 Evaluation
- `popularity.py`: the base code for the popularity-based model that will be introduced in section 4 Extension
- `sampling.py`: the code used for downsampling the training set that will be introduced in section 2.1 Downsampling

In the `test` folder, we have files to test the performance of the model with the optimal hyper parameters.

2 IMPLEMENTATION

2.1 Downsampling

For the efficient modeling and debugging process, I built the downsampled training set from 1%, 5%, and 25% of `cf_train_new.parquet`. The downsampled data needed to include enough users from the validation set to test my models. Therefore, I extracted all interactions of users in the validation set and combined with the random samples from the rest.

```
user_id_val = spark.sql('select tr.* from train tr
                         inner join validation va on va.user_id = tr.user_id')
user_id_val_not = spark.sql('select * from train
                      where user_id not in (select distinct user_id from df_val)')
sample = user_id_val.union(user_id_val_not.sample(fraction=0.01, seed=3))
```

I got three downsampled traning sets: `cf_train_tiny.parquet` has 626,795 tuples, `cf_train_small.parquet` has 2,613,424 tuples, and `cf_train_medium.parquet` 12,556,507 tuples. The number of tuples was each 1.26%, 5.25%, 25.20% of the original training set, which was not the same as 1%, 5%, and 25%, but I assumed it was not a problem for modeling. These files were saved under my personal HDFS directory.

2.2 PySpark Modules

The following table shows the PySpark modules, classes, functions, and methods used for modeling.

Table 1: PySpark modules used

Name	Modules	Purpose
ALS	pyspark.ml.recommendation	Alternating Least Squares (ALS) matrix factorization
StringIndexer	pyspark.ml.feature	Map `user_id` and `track_id` to indexes
RankingMetrics	pyspark.mllib.evaluation	Compute mean Average Precision
SparkSession	pyspark.sql	Create a Spark session
Window	pyspark.sql	Aggregate the true `track_id` to the user level
col	pyspark.sql.functions	Aggregate the true `track_id` to the user level
expr	pyspark.sql.functions	Aggregate the true `track_id` to the user level

2.3 Modeling

I made the ALS model in the following steps. The driver-memory and executor-memory were both set to 4GB.

ALS Modeling

1. Create a Spark session

2. Read the training and the validation/test parquet files as an RDD
3. Create DataFrames from the RDD
4. Combine the training and validation/test DataFrames
5. Fit the two StringIndexer for `user_id` and `track_id` separately with the combined DataFrame
6. Transform the training and the validation/test DataFrames by applying the two fitted StringIndexers separately
7. Fit an ALS implicit feedback model with the training DataFrame
8. Create recommendations for all users through the ALS model
9. Transform recommendations to an RDD
10. Transform the true `track_id` in the validation/test set to an RDD
11. Combine the recommendations RDD and the true RDD
12. Compute the raking metrics (mean Average Precision) with the combined RDD

3 EVALUATION

3.1 Ranking Metrics

I used the mean Average Precision (mAP) of the top 500 recommended tracks as ranking metrics for both the validation and test sets. We can compute the mAP with `meanAveragePrecision` in PySpark's `RankingMetrics`.

3.2 Hyperparameters

I tried several values for the following hyperparameters.

- *rank*: the number of latent factors
- *maxIter*: the maximum number of iterations to run
- *regParam*: the regularization parameter in ALS
- *alpha*: the baseline confidence in preference observations

3.3 Validation Results

The following table shows the results of the primary validation models. In addition to the hyperparameters mentioned above, I changed the number of recommendations to users (# *Rec*) to make the evaluation process faster.

Table 2: ALS Validation results

	# Rec	Training Size	Rank	maxIter	regParam	alpha	mAP	UpTime	Note
1	10	25%	20	20	0.01	2	0.020	6.5h	
2	10	100%	20	20	0.01	2	0.024	8.8h	
3	50	100%	20	20	1	1	0.026	21.1h	
4	500	25%	20	20	0.01	2	0.033	33.6h	
5	500	100%	50	20	1	0.1	N/A	N/A	Killed to free up resources
6	500	100%	50	20	1	10	0.052	58.0h	
7	500	100%	50	20	0.01	2	N/A	N/A	Killed to free up resources
8	500	100%	100	20	0.01	2	N/A	N/A	Killed to free up resources
9	500	100%	50	20	1	2	N/A	N/A	Killed to free up resources
10	500	100%	50	20	10	2	N/A	N/A	Spent 50 hours but not complete

The mean Average Precision of model 6 was 0.052, which looked like a good candidate for the optimal model. However, I could not get enough results for other hyperparameters (model 5, 7 -10) mainly because they needed a long time (more than 50 hours) to finish the evaluation. Fitting ALS was successful in those models, but the bottleneck was to extract the track indexes from the output (track index, score) of `recommendforAllUsers()` and transform

them to the RDD. This was a necessary preparation to use `RankingMetrics`, but in particular, the second and third lines of the following code took a lot of time to complete in proportion to the number of recommendations to users.

```
userRecs = model.recommendForAllUsers(numItems=500)
rdd_pred = userRecs.rdd.map(lambda x: (x.userIndex, [i[0] for i in x.recommendations])
```

Table 3: Examples of the bottleneck transformation

	userRecs		rdd_pred
userIndex	recommendations (trackIndex, score)	userIndex	Recommendations (trackIndex)
463	[[6835, 0.48], [1368, 0.33], [4674, 0.21], …]	463	[6835, 1368, 4674, …]
7253	[[2290, 0.98], [536, 0.74], [3327, 0.60], …]	7253	[2290, 536, 3327, …]
6654	[[50, 0.73], [4982, 0.54], [4648, 0.42], …]	6654	[50, 4982, 4686, …]

3.4 Test Result

As I could not solve the bottleneck mentioned above within the project timeline, it was impossible to make an optimized model for the test set. Therefore, I decided to make a sub-optimal model with 50 recommendations to all users as a baseline. The below table shows the result of the sub-optimal model. The mean Average Precision of the same model with the validation set was 0.026; however, the one with the test set slightly increased to 0.027.

Table 4: ALS test result

	# Rec	Training Size	Rank	maxIter	regParam	alpha	mAP	UpTime	Note
1	50	100%	20	20	1	1	0.027	6.7h	

4 EXTENSION

As an extension, I implemented a popularity-based baseline model manipulating the DataFrames. For local implementation, I additionally created the tiny training set that was 0.01% of the original training set. I made the model in the following steps. The results of the validation set and test set are shown in Tables 5 and 6.

Popularity-based Modeling

1. Same as the ALS Modeling step 1 - 5
2. Compute the average count μ over all interactions by the total interactions $|R|$ and a dumping factor β_g
3. Compute the average user difference b_i for each track by μ, $|R[:, i]|$, and a dumping factor β_i
4. Compute the average track difference b_u for each user by μ, b_i, $|R[u]|$, and a dumping factor β_u
5. Create recommendations for all users by sorting tracks by descending $\mu + b_i + b_u$ (equivalently b_i)
6. Transform recommendations to an RDD
7. Transform the true `track_id` in the validation set to an RDD
8. Combine the recommendations RDD and the true RDD
9. Compute the raking metrics (mean Average Precision) with the combined RDD

Table 5: Popularity-based validation results

#	# Rec	Training Size	β_g	β_i	β_u	mAP	UpTime	Note
1	10	0.01%	1	1	1	2.4E-4	N/A	Tested in a local environment
2	10	0.01%	1	10	1	3.8E-4	N/A	Tested in a local environment
3	10	0.01%	1	50	1	5.4E-4	N/A	Tested in a local environment
4	10	0.01%	10	1	1	2.4E-4	N/A	Tested in a local environment
5	10	0.01%	10	10	1	3.8E-4	N/A	Tested in a local environment

#	# Rec	Training Size	β_g	β_i	β_u	mAP	UpTime	Note
6	10	0.01%	10	50	1	5.5E-4	N/A	Tested in a local environment
7	500	100%	1	1	1	1.3E-6	21h	
8	500	100%	1	10	1	1.5E-5	21h	

Similar to the ALS model, the long execution time (more than 21 hours) made it difficult to tune the hyperparameter β_g, β_i, and β_u. Therefore, I decided to make a sub-optimal model with 50 recommendations to all users as a baseline. The below table shows the result of the sub-optimal model. The mean Average Precision was much lower than the ALS model, indicating that more precise parameter tuning and more recommendations were required to improve the performance.

Table 6: Popularity-based test results

#	# Rec	Training Size	β_g	β_i	β_u	mAP	UpTime	Note
	50	100%	1	1	1	5.83E-6	29.2h	

5 POINTS OF IMPROVEMENT

The biggest improvement point of this project is to modify the approach to extract the track indexes from the output of `recommendforAllUsers()`. This would be improved by more efficient parallelization and partitioning. Also, I should have joined a team to make the debugging process much efficient and increase the scalability.

REFERENCE

[1] Robert Bell and Yehuda Koren. 2008. Scalable Collaborative Filtering with Jointly Derived Neighborhood Interpolation Weights. IEEE. https://ieeexplore.ieee.org/document/4470228

[2] Yehyda Koren. 2008. Factorization meets the neighborhood: a multifaceted collaborative filtering model. ACM, Inc. https://dl.acm.org/doi/10.1145/1401890.1401944

[3] Yifan Hu, Yuhuda Koren, and Chris Volinsky. 2009. Collaborative Filtering for Implicit Feedback Datasets. IEEE. https://ieeexplore.ieee.org/abstract/document/4781121