Final project. Music Recommender System

MAKSIM EREMEEV (MAE9785), TAOTAO TAN (TT1922), and XIAO QUAN (XQ264)*

1 INTRODUCTION

We successfully implemented a collaborative-filter based music recommender system that is trained over the entire Million Song Dataset [2] provided by Professor Brian Mcfee, hosted on NYU's HPC network. We used Mean Average Precision for 500 recommendations per user to evaluate our model's performance and achieved a mean average precision score of 0.051. In addition, we implemented two extensions to our model. One is a popularity-based baseline model. The other is a cold-start model that uses the timbre features contained in the Million Song Dataset. We compared and summarized the performance of both models to the collaborative-filtering based models.

2 IMPLEMENTATION OVERVIEW

2.1 Environment setup

As we needed to execute Spark scripts using custom libraries and frameworks, we stick to using `miniconda` environment rather than pre-installed `python` on Peel. Refer to `install_miniconda.sh` and `shell_setup.sh` scripts for details. We only use pip-delivered packages that can be found in `requirements.txt` file. For all runs, we requested 8GB of RAM per executor, with all other spark configuration parameters set to their default values. We also wrap our code in a python package by using the `setuptool` framework. That allows us to have a complicated code structure rather than a single script. This makes our code readable and easily scalable. One can build and install the package with the following commands:

```
python setup.py build
pip install .
```

The `recommender_system` package will be installed for the current miniconda environment then.

2.2 Code structure

The code structure is presented in figure 1. We try to split the complex logic of the experiments across different files. Moreover, we decouple scripts that have to be executed via `spark-submit` and logic by creating class files. In particular, `linear_regression.py` is responsible for fitting several linear regressions for the cold start, and `als_model_custom.py`

Spark scripts are python scripts that accept a number of parameters, perform operations, and then dump results in the HDFS. In particular, `preprocess_data.py` executes preprocessing logic for the entire dataset, `downsample.py` downsamles the preprocessed data to get a small chunk for testing, `downsample_with_item_ho.py` does the same but with some items constantly missing (for cold start), `convert_h5py_to_parquet.py` takes care of correcting the base features formats that we get from the 3rd-party model (cold start), `als_train_evaluate.py` trains and evaluates the ALS model given the hyperparameters, `train_linear_regression.py` fits several regression models to approximate latent representations for ALS (for cold start), `als_cs_train_evaluate.py` trains and evaluates the model with cold start, `popularity_train.py` trains and evaluates the popularity baseline, `factorization_train.py` trains and evaluates the matrix factorization baseline (see section 4).

*All authors contributed equally to this project.

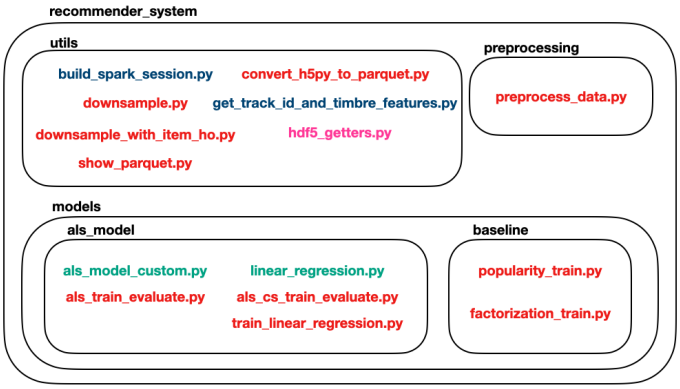

Fig. 1. The structure of the package. We highlight the spark scripts with **red** , class files in **green** , auxiliary scripts in blue , and 3rd-party code in pink.

2.3 Data-Preprocessing

Since we will be using the ALS model from the pyspark's ml.recommendation module, user ids, and the track ids need to be converted to integers for the model to process. To accomplish this, we extracted and unionized all unique user ids and track ids across the provided train, validation, and test datasets. We then provided each track and user id a unique integer index. We then inner-joined the index-ed ids to their original Dataframes. In this way, we have obtained unique integer track and user ids for training the model. During the experiments we found that repartitioning to 1000 blocks improves the efficiency of the pipeline.

2.4 Sub-sampling of Data For Local Prototyping

We first subsampled 1%, 5%, 10%, and 25% of the dataset for local model prototyping, using the `pyspark.sql.DataFrame.sample` method. We use fixed `seed = 12345` and sample without replacement.

2.5 Model Prototyping

Following the ALS paper [4] and pyspark implementation of `pyspark.ml.recommendation.ALS`, we setup our initial model with the following parameters:

```
rank = 10; max_iter = 8; regParam = 0.01; implicitPrefs = True;
alpha = 1.0; nonnegative = True; coldStartStrategy = `drop'
```

After seeing that the model succeeds in training and predicting using the sub-sampled datasets, we refactored and modulized our projects for more effective training and debugging.

3 EVALUATION

3.1 Spark<3.0 issue

During the experiments, we discovered that we cannot dump a trained model instance to HDFS, and load it back for evaluation purposes. Spark 2.4, which is installed on Peel, does not preserve the number of partitions for `itemFactors` and `userFactors` matrices when dumping the `ALSModel` instance. Therefore, a loaded-from-hdfs model instance's matrices always have a default number of partitions, making inference very slow (about 6h for the 25% dataset).

3.2 Results

We tune the ALS baseline by changing the `rank` and `regParam` parameters only. We used the complete training and validation set for parameter tuning (we do that manually by running the corresponding spark script). The validation results can be found in table 1. We report mAP and NDCG (Normalized Discounted cumulative Gain [5]) scores. We choose these scores since the former does depend on the order of the prediction, and the latter does not. Therefore, we believe comparing models on different

rank	regParam	mAP	NDCG@10	NDCG@50	NDCG@100	NCDG@500
10	0.01	0.0326	0.0521	0.0891	0.0991	0.1374
10	0.1	0.0322	0.0554	0.0892	0.1005	0.1395
10	1.0	0.0240	0.0421	0.0684	0.0872	0.1274
50	0.01	0.0463	0.0651	0.0963	0.1167	0.1759
50	0.1	0.0476	0.0672	0.1077	0.1181	0.1793
50	1.0	0.0390	0.0517	0.0798	0.1027	0.1717
100	0.01	0.0505	0.7942	0.1195	0.1373	0.1903
100	0.1	**0.05129**	**0.0834**	**0.1258**	**0.1461**	**0.1934**
100	1.0	0.0452	0.0682	0.1047	0.1256	0.1846

Table 1. Results for the ALS model given different hyperparameter values

However, as it can be seen from the table, all reported metrics correlate well. Here we discovered that model with `rank = 100` and `regParam = 0.1` performs best, and we use it to evaluate on the test dataset.

Finally, we ran the validation on the test data, achieving mAP of **0.05031** and NDCG@500 of **0.1895**.

4 EXTENSION 1. POPULARITY BASELINE-MODEL

4.1 Baseline: popularity

We first tried the most simplistic baseline: recommend items by averaging the utility. This is calculated by averaging the number of counts for all the users, and mathematically: $\sum_u R[u,i]/|R[:,i]|$. In Spark, we implement this calculation by averaging the count, group by tracks. The dataframe was then sorted by the average count. Then we recommended the top 500 tracks for each user. The result was evaluated using MAP (mean average precision [1]). We used the entire dataset for the evaluation. For the entire dataset (test set), we get mAP of 9.3×10^{-7}.

4.2 Baseline: matrix factorization

We tried to factorize the utility matrix into a global mean, an item vector, and a user vector. To factorization the utility matrix, several steps have been taken:

1. Calculate the average mean (μ).

2. We then calculated a vector b_u, which represents the average item difference from μ for user u. Think of this utility matrix as a sparse matrix, with u rows (users) and i columns (items). The vector is calculated by taking a row sum, and the results are then subtracted by μ. This number is finally normalized by the number of items for that user, and optionally, we can add a damping factor β_1. Here we use $\beta_1 = 5$. In math formula, this step could be described as $b_u = (\sum_i (R[u,i] - \mu))/(|R[u,:]| + \beta_1)$

3. The next step is to calculate a vector b_i, which represents the average user difference from $\mu + b_u$ for item i. First, we take the column sum of the utility matrix. The results are then subtracted by $\mu + b_u$, and normalized by the sum of

the number of users for that item and a damping factor β_2. Mathematically, $b_i = (\sum_u (R[u,i] - \mu - b_u))/(|R[:,i]| + \beta_2)$. I use $\beta_2 = 5$ here.

4. Once we get vector b_i, we can sort this vector and recommend the top 500 tracks to every user.

Notice the procedure is slightly different from what we have learned in the class. Here we calculate the user vector b_u first, then item vector b_i. The interpretation might be slightly different, but the main idea preserves.

We get the 2.5×10^{-4} for the sub-sampled dataset, and get 4.2×10^{-6} for the full dataset. Both of them are slightly better than a popularity-based model.

We tried to make a prediction by taking the Cartesian join of the user vector and item vector. The rationale is that we can fill in all the missing values by adding up each element in b_i, each element in b_u, and the global mean μ. We soon encouraged problems because the calculation took an enormous amount of time to run. Then we realized that procedure of cross join is unnecessary, and the equivalent way (and more efficient) is to sort b_i, and recommend top tracks to all the users.

In summary, the ALS model significantly outperformed the baseline model. Within the baseline model, the factorization-based model is slightly better than the popularity-based model.

5 EXTENSION 2. COLD-START

For the cold-start extension, we divided the work into two parts. The first is to extract the supplementary features that are nested in the MSD dataset. The second is to build a similarity-based model using these features.

5.1 Collect Audio Features

We decided to use the track timbre features [3] included in the full MSD dataset for use in our model. These are essentially (? , 12) shaped matrices, with the first dimension representing variable-length time slices per track, and each of the 12 columns approximating low-level acoustic features, such as loudness and timbre. To save space and computation in the model, we decided to average across all time slices for each track, resulting in a (12,) shaped vector per track, representing their overall sound profiles. This will be used to compute similarity in our cold-start model. To access the .h5 files, the `pytables` package is necessarily installed. We used the `h5py` package for saving `.h5` files.

5.2 Learning the transformation matrix

Having the feature set, we train a matrix $W \in \mathcal{R}^{100\times 12}$, to approximate the ALS latent representations from the features. We trained 100 linear regressions on the training dataset features, each mapping the timbre representation into one component of the ALS latent item representation. We use `pyspark.ml.regression` package for these purposes. We use ElasticNet [6] version of regularization with regularization parameter equal to `0.1`.

5.3 Experiments

To test the cold-start model, we first remove 0.25% of items from the training set and fit an ALS model on top. We then predict the latent representation for the removed items with linear regressions and add them to the ALS items' matrix back. The cold-start model achieves mAP and NDCG@500 of **0.04172** and **0.1243** respectively on test set, while the baseline (with 25% items missing and `drop` cold start policy) makes it **0.04018** and **0.1221**. We see an improvement in both mAP and NDCG, but quite insignificant. Moreover, dropping 25% of items results in a significant drop in scores comparing to the standard ALS model discussed in section 3.

6 DIVISION OF LABOR

- Data Sub-sampling: Xiao Quan
- Model Prototyping: Xiao Quan, Taotao Tan
- Model Refactoring: Maksim Eremeev
- Model Evaluation/Hyper-Parameter Searching: Maksim Eremeev
- Baseline Model Implementation: Taotao Tan
- Cold start Model Data Preprocessing: Xiao Quan
- Cold start Model Implementation: Maksim Eremeev

REFERENCES

[1] Steven M. Beitzel, Eric C. Jensen, and Ophir Frieder. 2009. *MAP.* Springer US, Boston, MA, 1691–1692. https://doi.org/10.1007/978-0-387-39940-9_492

[2] Thierry Bertin-Mahieux, Daniel Ellis, Brian Whitman, and Paul Lamere. 2011. The Million Song Dataset. *Proceedings of the 12th International Conference on Music Information Retrieval (ISMIR 2011)*, 591–596.

[3] Md. Afzal Hossan, Sheeraz Memon, and Mark A Gregory. 2010. A novel approach for MFCC feature extraction. In *2010 4th International Conference on Signal Processing and Communication Systems*. 1–5. https://doi.org/10.1109/ICSPCS.2010.5709752

[4] Yehuda Koren, Robert Bell, and Chris Volinsky. 2009. Matrix Factorization Techniques for Recommender Systems. *Computer* 42, 8 (Aug. 2009), 30–37. https://doi.org/10.1109/MC.2009.263

[5] Yining Wang, Liwei Wang, Yuanzhi Li, Di He, Tie-Yan Liu, and Wei Chen. 2013. A Theoretical Analysis of NDCG Type Ranking Measures. arXiv:1304.6480 [cs.LG]

[6] Hui Zou and Trevor Hastie. 2005. Regularization and Variable Selection via the Elastic Net. *Journal of the Royal Statistical Society. Series B (Statistical Methodology)* 67, 2 (2005), 301–320. http://www.jstor.org/stable/3647580