Million Song Dataset Recommender System

DS-GA 1004: Prof. Brian McFee Spring 2021

Recommender Systers: Jenna Eubank, Harlan Hutton and Harshitha Palegar

1. INTRODUCTION

Recommender systems efficiently employ techniques that curate products and experiences that users will enjoy in the future. Good recommender systems practice personalization by using mechanisms such as feedback to make decisions unique to the user to predict if they will respond to potential items. Such techniques and systems are what make companies like Spotify, Amazon and Netflix so successful. Predicting and utilizing user preferences can drive business and draw in new users. Our team created a recommender system in Spark using the Million Song Dataset to create song recommendations based on the implicit feedback, play count.

2. DEVELOPMENT

2.1 Initial data processing

To begin constructing our recommender system, we pulled 1% of the data to use in the development and testing of our baseline model. Our sampler.py code creates randomly sampled datasets from train, test, and validation based on a specified percentage of the original files. We then used the StringIndexer function to encode the user IDs and track IDs. The same indexers were applied to the train, test, and validation files, standardizing the encoding across all datasets.

2.2 Hyperparameter tuning

Prior to implementing the ALS model on the full dataset, we used both Bayesian optimization and grid search on 10% of the data to select rank and regularization parameters. While both methods provided similar results, grid search was computationally intensive but was also able to run on larger datasets on the cluster without additional environment manipulation. We optimized toward play counts reasoning that the more accurate projection of the counts would coincide with accurate rankings of potential preferences.

2.3 Code optimization for full dataset

Moving to the full dataset, we implemented the same code but found that inefficiencies caused a very long run time. String indexing on the full dataset was computationally taxing so we extracted that process

by saving the indexed dataframes once on the cluster and importing those files for each new model run. Additionally, we repartitioned the dataframes by 2000 to make use of parallel computing. We included the following arguments when submitting our job: driver memory = 5g, executor memory = 5g, and executor cores = 10. Adding these configurations allowed our code to more efficiently run on the full dataset.

3. EVALUATION

We chose MAP as our evaluation metric to focus on because it treats our recommendations like a ranked list and emphasizes the importance of getting “correct” or relevant recommendations high on the list. Spark includes MAP in the built in RankingMetrics function and can be designed as below.

$$\frac{1}{M}\sum_{i=0}^{M-1}\frac{1}{|D_i|}\sum_{j=0}^{Q-1}\frac{rel_{D_i}(R_i(j))}{j+1}$$

3.1 Further hyperparameter tuning

The MAP estimate on our first model of the full dataset was 0.039139 using the initial hyperparameters (rank = 20, iterations = 10, alpha = 1.0, regularization = .01). However, this score indicated that there was still significant room for improvement. We re-ran our grid search with expanded ranges for rank and iterations. Since we were only seeing marginal improvements using these hyperparameters we also evaluated different values for alpha. In addition to the grid search, we ran our model using a variety of the parameters to evaluate each MAP estimate on the full dataset.

3.2 Final Results

Our highest MAP estimate when evaluating model performance was 0.072533. We noticed increasing rank helped significantly improve the metric, with the final model at 175. However as expected, increasing rank also increased computation time. The final optimal hyperparameters also included 20 iterations, an alpha of 20 and the regularization parameter at 0.1.

Table 1: Model tested hyperparameters and resulting MAP estimates

Rank	Iterations	Alpha	Regularization	MAP Estimate
20	10	1.0	0.1	0.039139
25	10	1.0	0.1	0.041167
30	20	1.0	0.1	0.042815

175	20	20	0.1	0.072533
175	15	15	0.1	0.071236

4. EXTENSIONS

4.1 LightFM model

To create a comparison for our ALS model, we applied a LightFM model which analyzes 1% of the data (can be found on the other branch in the same repository). Preprocessing tasks included creating CSV files of the data after string indexing and transforming to Pandas dataframes. Because the LightFM model takes sparse matrices as its inputs, we created a user-item matrix of the entire dataset using scipy.sparse's csr_matrix. Using LightFM's cross-validation method, we split the user-item matrix into 80% train, 10% validation, and 10% test. Table 2 below shows different MAP estimates for different rank and regularization parameters with the best MAP being 0.000292. In terms of speed, hyperparameter tuning took 869.107 seconds and fitting of the final model took 35.086 seconds. We then used the same hyperparameters on the ALS model to compare runtime and MAP. From table 3, we can see both models have similar MAP estimates with ALS being slightly higher at 0.000327 at its optimal. In terms of runtime, ALS outperformed LightFM as it took 6.52 seconds for the model to fit.

Table 2: MAP Estimates for LightFM model by tested parameters

Regularization Parameter	**5 Rank MAP**	**10 Rank MAP**	**20 Rank MAP**	**40 Rank MAP**	**80 Rank MAP**	**160 Rank MAP**
0.01	0.000278	0.000290	0.000292	0.000290	0.000292	0.000290
0.1	0.000290	0.000290	0.000290	0.000289	0.000291	0.000289
1	0.000290	0.000290	0.000290	0.000291	0.000290	0.000289
2	0.000290	0.000291	0.000290	0.000291	0.000290	0.000289

Table 3: MAP Estimates for ALS model by tested parameters

Regularization Parameter	**5 Rank MAP**	**10 Rank MAP**	**20 Rank MAP**	**40 Rank MAP**	**80 Rank MAP**	**160 Rank MAP**
0.01	0.000204	0.000176	0.000240	0.000327	0.000303	0.000325
0.1	0.000151	0.000204	0.000236	0.000223	0.000214	0.000277
1	0.000162	0.000220	0.000211	0.000184	0.000185	0.000235

2	0.000159	0.000226	0.000222	0.000179	0.000192	0.000247

4.2 T-SNE visualizations

For our second extension we implemented T-SNE visualizations using an adapted version of code developed by Laurens van der Maaten. Our focus was to visualize the similarity of artists according to their genre tags. The primary challenge we faced was the high number of artists and terms for each artist which makes identifying distinct characteristics difficult. Even though T-SNE does do PCA to reduce dimensionality we also manually reduced the number of features by analyzing only the top 200 most frequently used terms. After hot encoding each artist for the top 200 genres we found that on average each artist had 35 associated terms and as many as 69. Our first visualization (Figure 1), which shows the similarities of the artists by term, without any identifying labels does not show meaningful clustering of artists. However, when we labeled artists by their primary decade (the decade where they released the most songs) trends emerged (Figure 2). By sampling smaller subsets of the data by filtering for artists who had more than 25 and fewer than 50 genre tags and taking a random sample of 1,000 artists we were able to see more meaningful patterns. We explored other visualizations including labeling for top genres, scaling, and further reducing the number of genres analyzed without meaningful results.

Figure 1

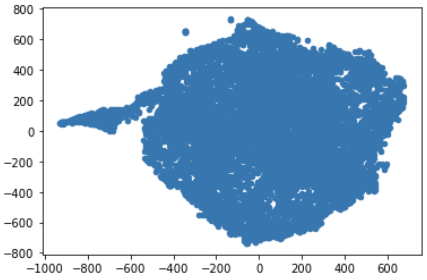

Figure 2

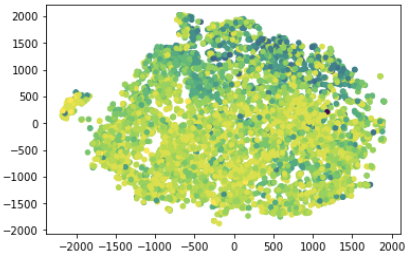

Figure 3

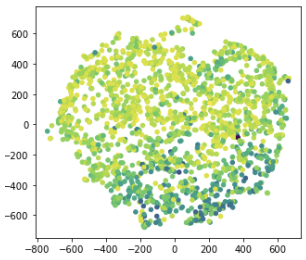

Figure 1: Artist genre similarities for 11,052 artists
Figure 2: Artist genre similarities for 6,058 artists labeled by decade (dark to light, old to new)
Figure 3: Artist genre similarities for 1,000 randomly sampled artists labeled by decade (dark to light, old to new)

Contributions

Jenna Eubank: Model development, extension 2, report
Harlan Hutton: Model development, extension 1, report
Harshitha Palegar: Model development, extension 1, report