Music Recommender Systems Using Implicit Feedback

Team Pseudo Recommenders
DS-GA 1004 Final Report
New York University Center for Data Science

Sudharsan Asaithambi	Aniruddha Chauhan	Pranab Islam	Sanjay Subramanian
sa6149@nyu.edu	ac8826@nyu.edu	pfi203@nyu.edu	ss14383@nyu.edu

Abstract

In this project, we implement collaborative filtering techniques in order to construct a recommender system for the Million Song Dataset[1]. Utilizing the alternating least squares method in Spark with an implicit feedback modeling structure, we learn latent factor representations for users and tracks in the dataset in order to provide 500 recommendations for each user. We compare our results to a baseline popularity model, use our learned representation of users and tracks to construct two-dimensional visualizations using t-SNE and UMAP, and compare our distributed computing implementation of ALS to a single-machine implementation. We produce a model with a mean average precision of 0.07982.

Introduction

The interaction data in the Million Song Dataset consists of implicit feedback – play count data for approximately one million users. The approach used here treats the data as representations of the strength of user actions (play counts). The model can then attempt to find latent factors which can be used to predict the expected preference of a user for a track based on the confidence of the observed user preferences.

The alternating least squares (ALS) algorithm factorizes the utility matrix R into two factor matrices U (user) and V (item/track) such that $R \approx U^T V$. The following minimization problem is subsequently solved[2]:

$$\underset{U,V}{argmin} \sum_{\{i,j \in \Omega\}} c_{i,j}\left(p_{i,j} - U_i^T V_j\right)^2 + \lambda\left(\sum_i \|U_i\|^2 + \sum_j \|V_j\|^2\right)$$

$$where\ p_{i,j} = \begin{cases} 1 & R_{i,j} > 0 \\ 0 & R_{i,j} = 0 \end{cases}, c_{i,j} = 1 + \alpha R_{i,j}$$

Here, we include a regularization term λ and a variable term $c_{i,j}$ which is correlated with modified implicit feedback values ($p_{i,j}$) through a constant parameter α. In order to maximize model performance, we tune these parameters and rank on a validation set before making 500 predictions for each user and evaluating ranking metrics on the test set.

Model Implementation

Data Pre-processing

The data included in the MSD contains full histories for approximately 1,000,000 users (training) and partial histories for 110,000 users (10,000 validation, 100,000 test). Each of these files contains tuples of (user_id, count, track_id), indicating how many times (if any) a user listened to a specific track. In order to make the

[1] Thierry Bertin-Mahieux, Daniel P.W. Ellis, Brian Whitman, and Paul Lamere. The Million Song Dataset. In Proceedings of the 12th International Society for Music Information Retrieval Conference (ISMIR 2011), 2011
[2] https://ci.apache.org/projects/flink/flink-docs-release-1.3/dev/libs/ml/als.html

data readable for the ALS implementation, we used the StringIndexer class of Spark to transform the user_id and track_id columns of our dataframes into integers. We chose to not down-sample the datasets when running our final model.

Model Training / Evaluation Pipeline

After data was pre-processed, an ALS grid was trained on the training set. The hyperparameter setting for each model was chosen based on a series of parameter grids that were employed. Initially having a wider scope for hyperparameters, the search was eventually narrowed as it became evident which hyperparameters were best. We trained our model serially on the cluster for ranks 10 to 250, regularization parameters 0.01 to 10, and alpha values from 10 to 150. We predicted for 5 users initially to have faster runs while doing a broad hyperparameter search. In every iteration, the coldStartStrategy parameter was set to "drop" so that predictions on the validation and test sets skipped over users who were not seen in the training set.

Once a model was trained, it was either stored in the Hadoop File System or it was evaluated right after training while the model was still held in memory.

Spark job configurations were specified during run time in the terminal for the most part; this mainly consisted of altering driver memory, executor memory, and sometimes the number of cores.

Evaluation

Model evaluation was based on two standard ranking-based evaluation metrics: mean average precision (MAP) and normalized discounted cumulative gain (NDCG). Each ranking system deals with a set of users (M), a set of ground truth relevant documents (D), and a list of recommended documents ordered by decreasing relevance (R)[3].

We have decided to utilize both MAP and NDCG as our evaluation metrics, as they both represent the inclusion of all relevant documents in a query with order considered. We optimized our models for MAP when utilizing Spark for general hyperparameter tuning. For our single-machine extension, we used NDCG as it was the main available ranking metric in lenskit, and accordingly tracked NDCG while using Spark.

Mean Average Precision (MAP) A measure of how many of the recommended tracks exist in the set of true relevant tracks – in this metric, recommendation order matters, and penalty is assigned for relevant missing items.

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

Normalized Discounted Cumulative Gain (NDCG) A measure of how many of the first k relevant tracks exist in the set of true relevant tracks averaged across all users – unlike with precision at k, order matters here.

$$NDCG(k) = \frac{1}{M}\sum_{i=0}^{M-1}\frac{1}{IDCG(D_i,k)}\sum_{j=0}^{\mathrm{n}-1}\frac{rel_{D_i}\left(R_i(j)\right)}{\ln(j+1)},$$

[3] https://spark.apache.org/docs/2.3.0/mllib-evaluation-metrics.html#ranking-systems

$$n = \min(max(|R_i|, |D_i|), k), \qquad IDCG(D,k) = \sum_{j=0}^{\min(|D|,k)-1} \frac{1}{\ln(j+1)}$$

Results

Optimizing for MAP, the model that achieved the best score on the validation set had hyperparameters: rank = 250, regularization parameter = 0.1, alpha = 100. The metrics for both the validation and test sets are shown below for our best model as well as the performance of some similarly tuned models.

TABLE 1:

	MAP	**NDCG at k=500**
Best Model Validation	0.08077	0.27487
Best Model Test	0.07982	0.26756
Rank 200, RegP 1, Alpha 100 Model Validation	0.07836	*N/A*
Rank 200, RegP 10, Alpha 100 Model Validation	0.07836	*N/A*
Rank 250, RegP 1, Alpha 100 Model Validation	0.08077[4]	*N/A*

Extensions

t-SNE Exploration

While training the ALS models, we learn feature embeddings of the users and tracks of dimension equal to the rank of the respective U and V factor matrices. It is impossible to visualize vectors in a 200+ dimension space; t-SNE and UMAP are dimensionality reduction techniques that preserve some notion of location from high-dimensional to low-dimensional space. t-SNE and UMAP are particularly important in use cases like ours where a high dimensional space has many different but related low-dimensional manifolds, like those between our tracks and genres.

Software packages we used: sklearn, MulticoreTSNE, and UMAP-learn. We used MulticoreTSNE because of its optimization for large datasets as opposed to sklearn's TSNE.

We first mapped the track_id to the genres (tags) by using the last_fm database which contains 3 tables: tids, tags, and tid_tag. We retrieved the mapping of integers to relevant genres and then collected the relevant tracks and genres in the tid_tag table. Following that, we extracted the high dimensional item factor matrix generated by our ALS model of rank 250. The item factor matrix had access to only a hash of track_id so we had to reverse hash it to obtain track_id. Next, we joined the tid_tag table with the item factor data and fit the feature embeddings to t-SNE and UMAP to get the two-dimensional representation of the user factor embedding.

In the representations of top twenty genres, we can see that the most popular genres have a wide coverage and overlap, which indicates that there may be mislabeling in the data. There are instances in which a track has genre / tags such as "death metal" and "christian", which appear contradictory for a single track to have. To analyze the correlation between the item factors and genres, we plotted the tracks with their genres and saw clusters being formed. This can be seen in Figure 1, where clusters of the "country", "rnb", and "House" genres / tags are visualized. These genres / tags were chosen because they are perceived as very distinct from each other culturally. There is also some noise which is expected and is at least partially due to some tracks having tags that are culturally inconsistent with each other.

[4] The difference in validation MAP scores between this model and the best model was about 1e-9.

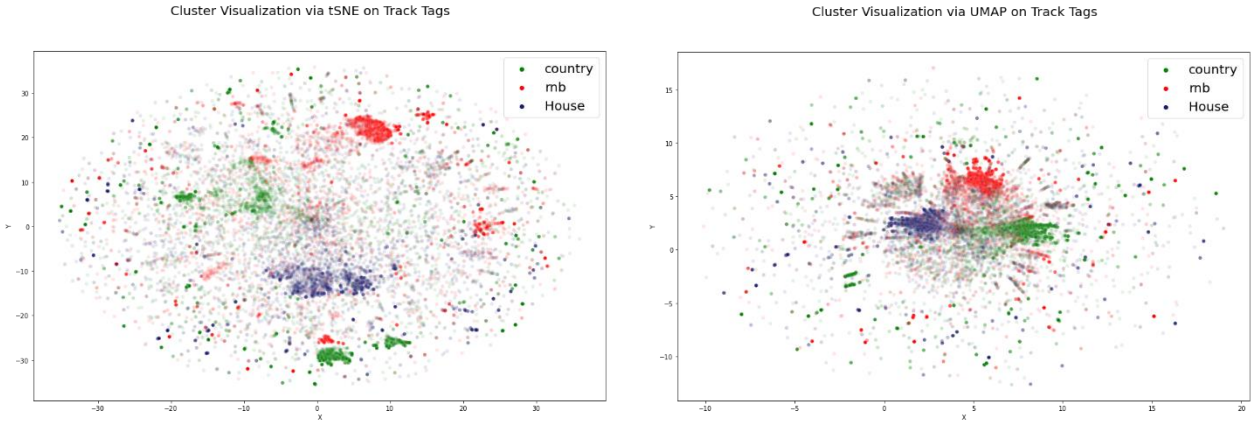

Figure 1

Popularity Baseline

For most recommender systems, comparison to some baseline popularity model is suggested, as these benchmarks can provide a reasonably high level of basic predictive power. In this instance, we implemented two versions of the baseline. The first version takes the absolute total count for each track in the training dataset across all users and selects the top 500. These tracks are used as predictions for every user in the test dataset. While in an implicit feedback model, counts may serve as an indication of user preference, this unbiased baseline is susceptible to examples with only one count, which may indicate negative sentiment.

The second version takes the average of track counts across all users, with an added damping term of 100 intended to offset instability in estimates for tracks with exceptionally low counts. As we chose not to drop tracks with low counts in our initial data pre-processing, we decided to implement this pseudo-count methodology for this baseline rather than dropping low-count tracks for consistency. This method potentially captures tracks which reached a broader scope of users (as opposed to the first method, where a small subset of users could skew a single track's counts exceedingly high). The metrics for these baseline models are reported below:

TABLE 2:

	MAP	**NDCG at k=500**
Method One	0.00263	0.01459
Method Two	0.00032	0.00283

In both cases, our best model outperformed the baselines by over one order of magnitude.

Comparison to single-machine implementations

We compared the performance of a single-machine implementation of ImplicitALS through lenskit with that of Spark's ALS. We trained both models with the same parameters and noted the difference in performance and time. The single machine implementation using lenskit was significantly faster than the implementation through Spark ALS. The entire lenskit model was fit within ten minutes and predictions were made for every user within approximately two hours. For Spark ALS, fitting a model and making predictions could easily take multiple hours unless one maximizes parallelism, memory, and core usage. Without maximizing those configurations, Spark ALS took an order of magnitude longer.

TABLE 3:

	Time Taken	**NDCG at k=500**
Lenskit	10 min fit, 2 hrs predict	0.30407
Spark ALS	~2.8 Hrs fit and predict[5]	0.26756

[5] Utilizing notably high resources for the Spark job (7 executor instances, default parallelism of 40, 4 cores, and 8gb of executor and driver memory). When less resources are used, run time could be an order of magnitude higher.

Contributions

Sudharsan Asaithambi – Initial ALS working pipeline, hyperparameter tuning, single-machine extension

Aniruddha Chauhan – Extensive hyperparameter tuning, full ALS pipeline, exploration extension, single-machine extension

Pranab Islam – Data pre-processing, initial ALS working pipeline, exploration extension

Sanjay Subramanian – Data pre-processing, initial ALS working pipeline, baseline extension