ALS Recommendation System on Million Song Dataset

WEI-LUN HUANG

New York University, Center for Data Science, wh2103@nyu.edu

Charlotte Ji

New York University, Center for Data Science, tj810@nyu.edu

Hengjiali Xu

New York University, Center for Data Science, hx2058@nyu.edu

We apply Spark's Alternating Least Squares (ALS) Matrix Factorization for data with implicit feedback (MF-Implicit) to tackle the Million Song Dataset (MSD) challenge, a learning-to-rank problem whose data only contains implicit user listening counts instead of explicit ratings. For our base model, we tune the ALS hyper-parameters over the validation set. As our first extension and best model, we experiment with logarithmically smoothed counts and conclude that it consistently outperforms the base one under MAP/NDCG on top 500 recommendations. For our second extension, we conduct experiments to accelerate search at query time and find significant gains using Scalable Nearest Neighbors (ScaNN) and Approximate Nearest Neighbors (Annoy) over brute force. For the third extension, we visualize the learned item latent factors with UMAP in two-dimensional space and observe that some genres have corresponding clusterings. Finally, we make additional efforts to handle the item cold-start problem by regressing item features to their latent factors from our best model.

Keywords: Million Songs Dataset, Recommender system, Matrix Factorization, Implicit feedback, Learning to rank, Alternating Least Squares

1 Introduction

Million Song Dataset (MSD) is the largest public dataset for offline evaluation of music recommender systems. The recommendation task is typically framed as a learning-to-rank problem that requires a model to predict a list of ordered music tracks for each user based on their preference levels.

It involves several modeling difficulties: 1. Data sparsity; 2. Implicit feedback; 3. A tremendous number of items. Since most users would regularly listen to only a few songs, especially for the popular ones, the data is highly sparse and may easily overfit by treating missing all entries as zeros. With only the implicit listening counts, we also have to adopt a modeling strategy apart from the common one for explicit user ratings. Besides, a large number of items can lead to a well-known Maximum-Inner-Product-Search (MIPS) problem, a k-nearest-neighbor search in the sense of cosine distance. For each user, it is too time-consuming to compute their preference scores on all items by a linear scan, so we have to consider a sublinear approach in practice.

We implement a recommender system that aims to tackle each of these challenges. We describe our dataset in the following section, and in section 3, we report the methods that we implemented and their respective results.

2 Dataset

2.1 Data Description

In this project, we mainly use the Million Song Dataset (MSD) [1]. This dataset contains 1,000,000 tracks, along with metadata such as artist information, acoustic features, etc. We mainly focus on the user interaction data from the Million Song Dataset Challenge [2], which contains approximately one million users. Each row of this dataset consists of `(user_id, track_id, count)`. The `count` represents the implicit feedback, meaning the play count of `track_id` for `user_id`. The interaction data has been partitioned into training, validation and test sets.

2.2 Subsampling

Since our original dataset is too big, we subsample the data before building the model to test and debug our implementation more efficiently. Specifically, we subsample 1%, 5%, and 25% of the training data, respectively, using PySpark's sample function. To help the code run efficiently on the cluster, we set `executor-memory=5g and driver-memory=5g`. We store the subsampled data as parquet files.

3 Methods and Results

3.1 Base ALS Model

To alleviate the data sparsity and overfitting issue, a Matrix Factorization (MF) model learns a low-rank approximation of the user-item relevance matrix by decomposing it into low-dimensional user/item latent factors and reconstructing the matrix with their inner products. To tackle the implicit feedback problem, [6] have proposed an MF variant, MF-Implicit, which binarizes the raw count numbers and uses them as user preference levels to weight each user/item pair for training the model.

For the base model of this project, we train an MF-Implicit on MSD with the ALS model in PySpark's ML library. The ALS model only works for integer `user_id` and `item_id`, whereas in our interaction dataset, both `user_id` and `track_id` are strings. To convert them into integer indicators, we use the StringIndexer transformer from `pyspark.ml.feature` and set `stringOrderType='alphabetAsc'` so that the indexation is according to the ascending alphabetical order. We perform this transformer on both the `user_id` and `track_id` columns by using a Pipeline. For subsampled data, the indexation is done as described. For the whole data, we first union the training, validation, and test sets, and then fit the Pipeline on this unioned data. We do this step in order to have the training, validation, and test sets all have the same indexation. Finally, we perform the transformation on the training, validation, and test sets separately, and save them to new parquet files.

We then fix PySpark's default `alpha=1` and `maxIter=10` and tune the best regularization/rank hyper-parameters on the validation set by Mean Average Precision (MAP) and NDCG scores. The results are reported in Table 1. We observe that increasing the rank (i.e., model complexity) can consistently improve both MAP and NDCG. In addition, `regParam=0.1` performs the best because `regParam=1` might overregulate the model, which discourages the model from fitting the data well. The best MAP/NDCG on validation set is 0.057775/0.214427, and the corresponding test MAP/NDCG is 0.057583/0.213822. The gap between validation results is subtle, so the model is considered well-regularized and generalizable.

Table 1: Validation MAP/NDCG on raw count with different hyper-parameters

rank	regParam	alpha	MAP	NDCG
10	0.01	1	0.03329	0.140565
10	0.1	1	0.03342	0.141767
10	1	1	0.025835	0.126938
40	0.01	1	0.043353	0.173279
40	0.1	1	0.044826	0.176385
40	1	1	0.037337	0.163833
100	0.01	1	0.050036	0.193993
100	0.1	1	0.052634	0.199482
100	1	1	0.046243	0.189392
200	0.01	1	0.054379	0.20715
200	0.1	1	**0.057775**	**0.214427**
200	1	1	0.052872	0.207819

3.2 ALS Model with Logarithmic Smoothing

To further improve the base model as our first extension, we follow the same setting in [6] to smooth each listening count by taking `log10(1+count/1e-8)`. We also experiment with the PySpark's default `alpha=1` and `alpha=40` mentioned in the paper. As shown in Table 2, the models with smoothed count outperform the original ones with the same `rank` and `regParam` since the distribution of listening counts is highly skewed and long-tailed. We also notice that the models with `alpha=40` are more accurate than `alpha=1`. Since each count is closer to the other after taking the logarithm, a larger alpha can help the model distinguish a bigger count from a smaller count. The best MAP/NDCG of the smoothed models is 0.083526/0.286694, and the corresponding test MAP/NDCG is 0.083543/0.285611, which leads to the best original model by an increase of 0.03 MAP and 0.07 NDCG. Similar to our base model, the gap between the validation and test scores is insignificant, which shows that the model is well-regularized and hence less likely to overfit.

Table 2: Validation MAP/NDCG on smoothed count with different hyper-parameters

rank	regParam	alpha	MAP	NDCG
10	0.01	1	0.036273	0.152033
10	0.1	1	0.036587	0.152896
10	1	1	0.034101	0.148746
40	0.01	1	0.05127	0.195606
40	0.1	1	0.052655	0.198305

40	1	1	0.052917	0.200382
100	0.01	1	0.059263	0.218356
100	0.1	1	0.061583	0.22275
100	1	1	0.066538	0.233785
200	0.01	1	0.064202	0.232991
200	0.1	1	0.067337	0.238505
200	1	1	0.076182	0.257086
10	0.01	40	0.034091	0.160431
10	0.1	40	0.034529	0.161282
10	1	40	0.034846	0.162003
40	0.01	40	0.057302	0.224936
40	0.1	40	0.057294	0.225181
40	1	40	0.057766	0.22624
100	0.01	40	0.072276	0.262515
100	0.1	40	0.072355	0.262796
100	1	40	0.072734	0.264045
200	0.01	40	0.083106	0.284668
200	0.1	40	0.083285	0.285173
200	1	40	**0.083526**	**0.286694**

3.3 Exploration

Our second extension is to generate a visualization of the items using the learned representation. Since we do not have explicit features for users in the data, we only develop visualizations of the tracks. Specifically, we use the learned latent representations of items from the ALS model with the hyperparameters that lead to the best validation evaluation (i.e., `rank=200, regParam=1, alpha=40`). For the feature we use to explore, we decide to use the first term of the artist in each track since it is the most representative for the artist. We find that there are 657 unique first terms among all tracks. Considering the interpretability of our visualization, we subsample 10,000 tracks from the data to reduce the number of terms, which leads to 472 unique terms. Then, we count the number of samples for each term and extract the ones that have the highest frequencies, such as pop music, rock. For some subgenres, we convert them into their corresponding main genres (e.g., “future rock” is converted into “rock”). We get a total of 22 terms and eliminate the tracks with any other terms.

Figure 1: UMAP embedding for items labeled by genre of their corresponding artists

We use UMAP to reduce the dimension of the learned latent representations from 200 to 2. As a nonlinear dimensionality reduction method, it does not require any pre-processing step such as PCA and can be used to sparse matrices directly. It is faster to train and can produce better visualization than t-SNE [4].

As shown in Figure 1, we can observe some clusterings of the tracks. The purple dots and green dots are the most obvious, which correspond to hip hop and metal music artists. This means that hip hop and metal music are well learned by our model. The other groups of dots seem to be scattered randomly on the graph, which could be due to the limit of our feature engineering algorithm. Specifically, the genre of the track might not exactly match the top genre of its artist.

3.4 Fast Search

Our third extension is to implement a spatial data structure to accelerate search at query time, that is, to tackle the Maximum Inner Product Search (MIPS) problem. Various existing methods aim to reduce the number of evaluated items by separating the latent vector space into spatial regions, including KD-Tree, BallTree, and Annoy [5]. More recently, another line of research focusing on further accelerating the inner-product operations with quantization and fixed-point operations, such as ScaNN [3], has been shown to outperform the former ones.

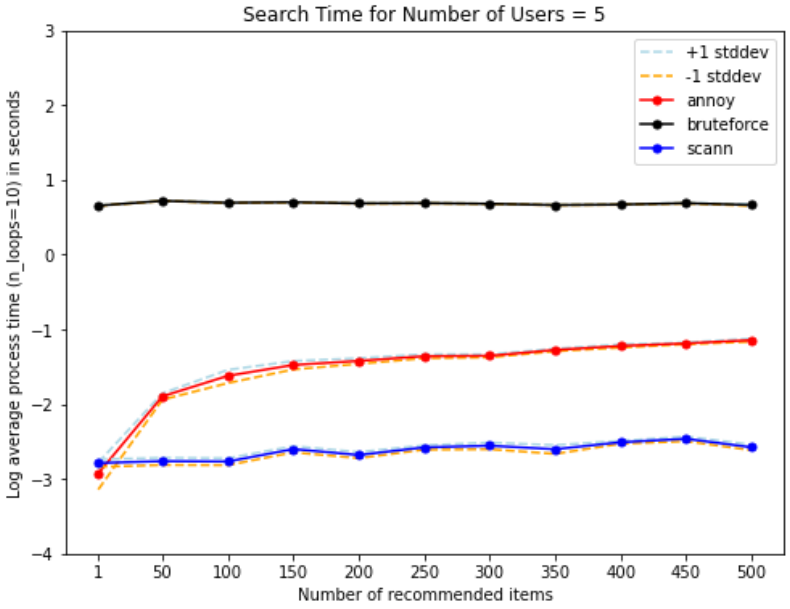

Figure 2: Log average time using Annoy, ScaNN and brute force for querying `k=1,50:500:50` items for 5 users

We compare the above two strategies for the MIPS problem. We choose Annoy and ScaNN as two respective representatives. We also implement a brute-force linear search as the baseline. We time queries in parallel on Greene by submitting one array job for each of the three methods. For each array job, we request 1 node, 1 task per node, and 1 CPU per task with 32GB memory, and we run all jobs on medium memory compute nodes for better comparison.

Our result shows that the brute-force method performs much worse than the other two overall due to poor linear time complexity. The running time of the brute-force one is almost independent of the number of recommended items `k`, which corresponds to the running time complexity `O(nd+k)=O(nd)` for `n>>k`, where `n` is the number of items and `d` is the dimension of latent factors. For the other two methods, ScaNN generally outperforms Annoy except for a very small `k`, which reflects the smaller time complexity of a quantized vector inner product `O(pd + mn)<<O(nd)` by using lookup tables, where `p` is the size of each quantization codebook and `m` is the number of codebooks. Since we adopt the default setting for the GloVe dataset [7] in their Github example[1], the configuration for the spatial data partitioning trees might not be suitable for a small `k` and would incur a larger overhead to ensure the high recall of searched items.

3.5 Bonus: Cold Start

[1] https://github.com/google-research/google-research/blob/master/scann/docs/example.ipynb

We implement multi-output ridge regression by wrapping SciKit-Learn's MultiOutputRegressor on Ridge to predict latent item vectors using track-level features. To do so, we extract features from the dataset, including release, artist_id, duration, loudness, and tempo, for all 385,371 tracks in the MSD training dataset. We perform one-hot encoding on release and artist_id using Dask-ML's OneHotEncoder and fill in missing values of duration, loudness, and tempo with their respective means using SciKit-Learn's SimpleImputer. This produces 112,513 features in total, including 112,510 binary variables that represent release and artist_id and the 3 continuous variables (duration, loudness, and tempo). Our target variables are 200 latent item vectors produced by fitting our best ALS model (MF-Implicit with `alpha=40, rank=200, regParam=1 and iteration=20`) on the MSD training data.

For the ridge regressors, we tune the regularization parameter on a 15% validation set. The best multi-output ridge regression model achieved an R-squared of 0.3905 and mean-squared error of 0.0007 on a 20% test set, measured by uniformly averaging over all regressors. We saved the model's weights fitted on the entire MSD training data.

We are not able to simulate the cold-start scenario in time due to the long wait time on Peel at the end of the semester. As future work, we would like to evaluate how the regression model affects our best ALS model.

CONTRIBUTIONS

Wei-Lun Huang worked on base ALS evaluation and ALS with smoothed labels on Peel's Spark cluster, and implemented fast search on a local laptop.

Charlotte Ji worked on subsampling datasets, extracting + engineering features and modeling for cold start, and parallelizing and plotting for fast search.

Hengjiali Xu worked on data preprocessing (indexation on user_id and track_id), building ALS model, and exploration parts.

All the members equally contributed to the final writeup, drafted their own implementation parts, and proofread others' sections.

REFERENCES

[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] McFee, B., Bertin-Mahieus, T., Ellis, D. P., & Lanckriet, G. R. (2012, April). The million song dataset challenge. In Proceedings of the 21st International Conference on World Wide Web (pp. 909-916).

[3] Guo, R., Philip Sun, Erik M. Lindgren, Quan Geng, David Simcha, Felix Chern and S. Kumar. "Accelerating Large-Scale Inference with Anisotropic Vector Quantization." ICML (2020).

[4] McInnes, L., & Healy, J. (2018). UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction. ArXiv e-prints.

[5] Li, W., Ying Zhang, Y. Sun, Wei Wang, W. Zhang and X. Lin. "Approximate Nearest Neighbor Search on High Dimensional Data — Experiments, Analyses, and Improvement." IEEE Transactions on Knowledge and Data Engineering 32 (2020): 1475-1488.

[6] Hu, Yifan, Y. Koren and C. Volinsky. "Collaborative Filtering for Implicit Feedback Datasets." 2008 Eighth IEEE International Conference on Data Mining (2008): 263-272.

[7] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. (2014). GloVe: Global Vectors for Word Representation.