Collaborative Filtering on the Million Song Dataset

DS-GA 1004 Big Data

Ben Berkman, bjb433@nyu.edu

Zixiao Chen, zc2194@nyu.edu

Stanley Sukanto, ss14358@nyu.edu

David Trakhtenberg, dt2229@nyu.edu

We build and evaluate a recommender system on a dataset of one million songs and one million user interactions (as well as metadata, audio features, and more). Given the nature of play count data, we utilize PySpark's implicit feedback model to train the model and perform hyperparameter tuning on the rank and regularization of the latent factors, maximum iterations of the Alternating Least Squares model, scaling parameter of the counts, and whether to use nonnegative constraints for least squares. Once we establish an initial ALS model, we implement two extensions: a popularity-based baseline model and a visualization of the items and users using the UMAP algorithm. We conclude with a discussion on next steps.

1 INTRODUCTION

To design the recommender system, we opt for a collaborative filtering model versus a content-based one. Collaborative filtering may suffer from a cold start problem whereas content models need little data to start but are far more limited in scope. We learn a factorization of a large, sparse data matrix R into a user factor matrix U and an item factor matrix V where $R \approx UV^T$. The data matrix R represents count data for each track by user, where a single entry $R_{i,j}$ represents the number of times user i listened to track j. A common approach is to use a neighborhood model, where given a user u we find the most similar users u′ and predict items v with high feedback between similar users. Ultimately, the goal is to predict the missing entries of R.

As our data contains implicit feedback (play counts) as opposed to explicit feedback (e.g., ratings), we implement an alternating least squares (ALS) algorithm whereby we fix user factors u to solve for track factors v. In fixing these latent factors, the algorithm can boost computation by achieving parallelism. As count data is challenging to predict, we instead estimate binary interactions (i.e., seen/not seen) and use counts to weight terms [1]. We aim to solve the following non-convex optimization problem:

$$min \sum_{(i,j)\in\Omega} c_{ij}(p_{ij} - < U_i, V_j >)^2 \quad \text{where} \quad p_{i,j} = \begin{cases} 1 \; if \; R_{ij} > 0 \\ 0 \; if \; R_{ij} = 0 \end{cases} \quad \text{and} \quad c_{i,j} = 1 + \alpha R_{ij}$$

where preference p is a binary indication of whether a user-item interaction exists. The confidence c in a preference p is linearly correlated with feedback values by a scaling factor α (i.e., weight per interaction). Therefore, there is a stronger confidence that user i prefers track j when a count $R_{i,j}$ is larger.

2 IMPLEMENTATION

2.1 DATASET OVERVIEW

The dataset contains one million songs and listening histories for one million users which was then oriented into three parquet files (*cf_train, cf_validation, cf_test).* Each row in the dataset represents one interaction between the user and item which consists of three columns: *user_id* (the specific user), *count* (play count of track), and *track_id* (the specific track).

2.2 Preprocessing Data

The dataset is large (*cf_train* has 49,824,519 records; 110,000 of which are unique users and 163,206 are unique tracks). We downsample the parquet files to 10% of the original dataset while we iterate on the best partition methods to read the parquet files into a SparkSession. After we repartition on *user_id* and *track_id*, we use PySpark's StringIndexer[1] to encode the string values of the dataset to integer values as PySpark's ALS module only supports integers. To ensure the user-item interactions in the validation set is accounted for when performing hyperparameter tuning, we select users and items to be present in both train as well as validation.

2.3 ALS Implementation

We fit PySpark's ALS model[2] on the string-indexed training dataset. The model is first fit on a subset of training data on a local machine, then run on NYU's High Performance Computing cluster to fit and evaluate the full dataset using parallel computation. A grid of hyperparameters is tested (see Section 3.2, as well as the "hyperparam-tracker" within the GitHub repository for full grid) against the validation dataset, before the final selection is evaluated against the test set.

To support reliable parallel computation, we use several configuration options. First, a checkpoint interval and directory are set to periodically save parts of the model and free storage space as the algorithm fits the remaining data. Second, blacklisting is disabled to allow executors that have failed in the past to try to continue running. Further improvements include efficiently partitioning the data (described above) and requesting five cores and 4 GB of driver memory. A strong model fits on the full training data and makes recommendations on the validation data in approximately one hour. The model saves to a folder, then recommends on the test data.

3 IMPLEMENTATION

After fitting the model, we evaluate the top 500 track (*item*) recommendations for each user through hyperparameter tuning the validation dataset. To do this, we select the distinct users in the validation dataset (*user*), get the tracks each of these users have listened to (*truth*), and then also recommend 500 tracks to these users (*recommendations*). We then map from a data frame of *user*, *truth*, *recommendations* to an RDD per PySpark's RankingMetrics[3] requirements.

3.1 Metrics

We use mean average precision (mAP) to compare the effectiveness of the 500 recommendations in relation to a ground truth set[4]. Using ground truth labels $D_i = \{d_0, d_1, .., .d_{N-1}\}$ and recommendations $R_i = \{r_0, r_1, .., .r_Q\}$ the relevance score is defined as:

$$rel_D = \begin{cases} 1\ if\ r \in D \\ 0\ otherwise \end{cases}$$

For M users, mAP is calculated as:

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

where *Q* represents the top recommended items for each user (e.g., *Q=500*). mAP represents how many of the recommended tracks are in the set of true relevant tracks. The order of recommendations is considered as there is a penalty for highly relevant tracks that are not recommended.

---

[1] https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.feature.StringIndexer.html
[2] https://spark.apache.org/docs/2.2.0/ml-collaborative-filtering.html
[3] https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.mllib.evaluation.RankingMetrics.html
[4] https://spark.apache.org/docs/2.2.0/mllib-evaluation-metrics.html

3.2 Hyperparameter Search and Best Model Select

We perform grid search on the following hyperparameters:

- Rank of latent users/tracks (Rank)
- Regularization parameter to control complexity (l)
- Maximum number of iterations (maxIter)
- Scaling parameter for handling count data (a)
- Non-negative constraints (Nonnegative)

We test 73 hyperparameter combinations to ensure optimal model performance. Largely we approach the search with a manual analysis on how different parameters influence the mAP metric and re-evaluate from there. For example, we quickly learn that increasing Rank or maxIter increases performance but also increases training time. Using the validation dataset and optimizing for mAP, the best model has $rank = 400,\ \lambda = .1,\ maxIter = 30,\ \alpha = 40,\ Nonnegative = True$ and achieves a mAP of 0.084020. Select results are shown in Table 1, and full results are found within the repository's "hyperparam-tracker.xlsx" file.

However, this "best" model requires more than 24 hours of training time due to its high rank and nonnegative least squares. Instead, we use the second highest performing model, which requires only one hour to train, to recommend on the test data and for the extensions. This second model ($rank = 100, \lambda = 0, maxIter = 20, \alpha = 20, Nonnegative = False$) achieves a mAP of .064908 on the validation dataset and .065080 on the test dataset.

Table 1: Hyperparameter tuning for select parameters

Rank	l	maxIter	a	Nonnegative	Validation mAP
400	.1	30	40	True	0.084020
100	0	20	20	False	0.064908
300	.1	70	20	True	0.042335
60	.1	10	10	True	0.030266
10	.01	20	20	True	0.006744

4 EXTENSIONS

We consider two extensions to this ALS model: comparing the ALS model to a popularity-based baseline model and exploring the learn representations to develop visualizations of the items and users.

4.1 Popularity-Based Baseline Model

When building our recommendation system, we decide to first test a popularity-based baseline model. This approach models each interaction as a combination of global, item, and user terms. Specifically, we have R[u,i] = μ + b[i] + b[u] where u represents the average rating over all interactions, *b[i]* represents the deviation of item *i* from μ, and *b[u]* represents the deviation of user u from μ [2]. Embedded within μ can be a damping parameter, β, which regresses items with fewer ratings towards zero. To recommend for user u, the model sorts items i by descending μ + b[i] + b[u] with μ + b[u] fixed for any given user. We implement this approach by hand, without use of external libraries such as lenskit. First we select the top 500 songs in the training dataset. We then collect the songs each user in the validation (or test) set has listened to. We recommend the top 500 songs from the training set to the users in the validation set, and calculate mAP. This approach yields a mAP of .01877 for the validation set, and .01990 for the test set.

To improve the baseline model, we add one limit to our predictions, which is that the recommended collection should not cover the songs that users already listened to. The rationale behind this limitation is that usually users wish to be recommended with new songs that they haven't listened to before. We first find the top 1000 most popular songs. Then we filter out songs that one has already listened to for each user. Lastly, we recommend the 500 most popular songs for each user, skipping songs that have already been consumed. This adjustment results in a mAP of .02138 for the validation set, and .02279 for the test set, which is higher than that of the original baseline model.

4.2 Visualization Exploration

For our second extension, we utilize the Uniform Manifold Approximation and Projection for Dimension Reduction (UMAP) [3] in order to visualize the learned representation of items. The genre information is obtained from the metadata available on the Million Song Dataset website. For visualization purposes, the dimension of the learned item representation is reduced from 100 to 2, as visualized in Figure 1. Since UMAP does not construct its representation by any meaningful distance metric, cluster sizes and distances between genre clusters carry little to no significance. Additionally, since UMAP is a stochastic algorithm, different runs yield different plots so tuning parameters like size of the local neighborhood and how tightly points are packed is a big consideration. However, we observe distinct clusters for each genre which signifies that items of the same genre are similar. Effectively, users receive recommendations for similar items in a given genre.

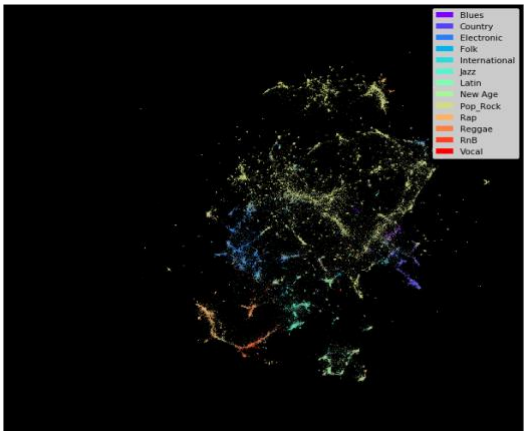

Figure 1: Low dimensional representation of learned user factors by genre

5 DISCUSSION AND NEXT STEPS

Recommender systems are highly prevalent throughout society, being applied in movies, music, shopping, etc. They influence large swaths of the human population every day therefore it is prudent that data scientists understand and account in their modeling for their impact. Due to a recommender's reliance on similarity, they tend to reduce the diversity of items that a user sees over time. Users eventually get trapped in a filter bubble [4], many times unknowingly. This can be harmful for several reasons: users feel isolated, communities become adversarial, and societies lose a sense of kinship. While there may not be a way to directly solve this, Kilitcioglu [5] offers a way for users to work their way out of their bubble via diversification. He informs that some systems attempt to keep diverse sets of interests amongst users naturally while other systems impute artificial diversity, even for users with specific interests. Pinterest data scientist Ahsan Ashraf gives the following measures to use as sanity checks for a system: stability (does the system converge to certain items), sensitivity (do the recommendations cover a spectrum of items as you artificially perturb the system), and sanity (are random recommendations more diverse than the system's recommendations). As it stands, the tools we have to make recommendations are far more advanced than our understanding of how to diversify those systems. As data scientists, it is our duty to ensure users and society are not manipulated by these systems to think a certain way and are given a diverse experience.

6 INDIVIDUAL CONTRIBUTIONS:

- Ben: ALS model, Extension 1 (Baseline Comparison), Hyperparameter Tuning
- Zixiao: Extension 1 (Baseline Comparison), Hyperparameter Tuning
- Stanley: ALS model, Extension 1 (Baseline Comparison), Extension 2 (Visualization), Hyperparameter Tuning
- David: ALS Model, Extension 2 (Visualization), Hyperparameter tuning

REFERENCES

[1] Yifan Hu, Yehuda Koren, and Chris Volinsky. "Collaborative Filtering for Implicit Feedback Datasets". In: 2008 Eighth IEEE International Conference on Data Mining (2008). DOI: 10. 1109/ICDM.2008.22.

[2] Francesco Ricci, Lior Rokach, and Bracha Shapira. 2015. Recommender Systems Handbook. Springer, New York, NY.

[3] McInnes, Leland, Healy, John and Melville, James UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction. (2018). , cite arxiv:1802.03426Comment: Reference implementation available at http://github.com/lmcinnes/umap

[4] Pariser, Eli. 2011. *The filter bubble: what the Internet is hiding from you.* London: Viking/Penguin Press.

[5] Kilitcioglu, Doruk. 2018. *Recommender Systems: From Filter Bubble to Serendipity*. https://dorukkilitcioglu.com/2018/10/09/recommender-filter-serendipity.html