DS-GA 1004 Big Data

Recommender System Project Writeup

Project team: Butterdogs

Pavel Gladkevich (pg2255)

Surabhi Ranjan (sr5768)

Ted Xie (tx607)

Duey Xu (dx2028)

May 2021

All code can be found at: https://github.com/nyu-big-data/final-project-butterdogs/tree/main

Introduction

Recommender systems are becoming integral in modern consumerism. The usage of these models can be seen implemented while browsing the internet for entertainment, shopping, and more. We are implementing our ALS (Alternating Least Squares) model on the Million Song Dataset (MSD) to showcase how music recommendations can be made, along with evaluation results, single machine implementation, popularity baseline model, and visualizations.

1. Implementation

1.1 Subsampling

The MSD was pre-split into training, validation, and testing sets. For our initial training evaluation and sub-sampling, we identified all the unique users present in the MSD validation dataset (10,000 users). We divided the train dataset into 2 tables - data for users present in validation and data for users not present in validation. To create samples of 1%, 5%, 25% and 50% of the entire dataset, we randomly sub-sampled from the first table and to create higher sub-samples (50% and above), we took the entire first table and the difference was sub-sampled randomly from the second table. This was done to ensure consistency and maximum overlap in the number of users we were recommending songs for across sampled datasets.

1.2 Implementation process

We implemented our model using PySpark's ALS recommendation module. The following steps were taken in order to get the ALS model working on the entire MSD dataset:

1. User IDs and Track IDs from all 3 datasets were hashed into integers using PySpark's hash function.
2. For the baseline model, we fit the ALS model onto the train dataset. We then predicted the top 500 songs for each user from the validation dataset. The list of songs that each user listened to in the validation dataset were our actual labels.
3. Joining the predicted song-list and the actual song-list on user ID, we created a list of lists where each element was a list of predicted song list and actual song list corresponding to a unique user.
4. We converted this list to a parallelizable intermediate Spark RDD for efficiency computation of our metrics - MAP and nDCG.

1.3 Hyperparameter tuning

Hyperparameter ranges were split across multiple parallel jobs to reduce search time. The following hyperparameters were tuned in our ALS model:

- **Rank**: Determines the complexity of the ALS model through the number of latent factors represented
- **Regularization**: Regularization parameter that penalizes complex model weight vectors

- **Alpha**: Determines the baseline confidence allocated towards preference observations

The hyperparameter "max iterations" which is an argument in the ALS function was not used, and a value of 10 was used for every execution of the algorithm. We also attempted to modify and use the grid searching function in the standard PySpark ALS pipeline, however the function was not well suited for our problem.

2. Evaluation

2.1 Metric used

We chose the Mean Average Precision (MAP) as the evaluation metric as precision identifies the percentage of the 500 recommended items that were relevant to each user and this is averaged over all users. Precision at k was determined to be an inappropriate metric for this project, given that the ground truth lengths for each listener differ considerably. For example, some users in the dataset have only listened to one song, while others may have listened to over 100.

2.2 Evaluation results

The following hyperparameter ranges were searched for this project.

- **Rank**: 5 to 310
- **Regularization**: 0.01 to 0.2
- **Alpha**: 1 to 100

A sample of our tuning results can be viewed in Appendix A-1. The following combination of hyperparameters produced the best results: **Rank: 205, Regularization: 0.01, Alpha: 100**. The corresponding MAPs on the validation and test data were **0.07916** and **0.07814** respectively**.** The most important hyperparameter for tuning was the rank, driving up our evaluation score. Regularization did not affect the MAP significantly, thus most iterations ran with regParam = 0.01. We found that a higher rate of increase in the confidence matrix, or alpha, worked better with larger subsamples.

3. Extensions

For our project we chose the single machine implementation, popularity baseline model, and visualizations using TSNE/UMAP as our extensions.

3.1 Extension 1: Single Machine Implementation (LensKit ALS)

We used LensKit to prepare a single machine implementation of the ALS algorithm. The hypothesis was that the time taken to complete each run would be significantly longer than through parallel cluster computing. Given that the single machine implementation on the entire dataset could take an inconveniently long amount of time, we also compared the run times on 1%, 5%, 25% and 50% subsamples of the data. Our run times and nDCG values from each run are below.[1]

	Computation approach	
Subsample size	Single machine (LensKit ALS)	Cluster computing (PySpark ALS)
1% training subsample	14 minutes 20 seconds nDCG = 0.038678	16 minutes 46 seconds[1] nDCG = 0.061231
5% training subsample	80 minutes 20 seconds nDCG = 0.066172	52 minutes 12 seconds[1] nDCG = 0.090670
25% training subsample	276 minutes 54 seconds nDCG = 0.145251	94 minutes 30 seconds[1] nDCG = 0.175929
50% training subsample	284 minutes 20 seconds nDCG = 0.19354099709630362	121 minutes 59 seconds[2] nDCG = 0.2220799264029042
Full training data	Execution timed out after 10 hours	151 minutes 29 seconds[3] nDCG = 0.2726371778842973

Unsurprisingly, we observed longer computation times when using a single machine. As the dataset grew larger, computation was faster on the cluster than on a single machine by 1.5 to 2.9 times. Our LensKit code is viewable in LK_run_v3.ipynb, while our PySpark ALS code can be found in ALS_cluster_dx.py.

3.2 Extension 2: Baseline Models

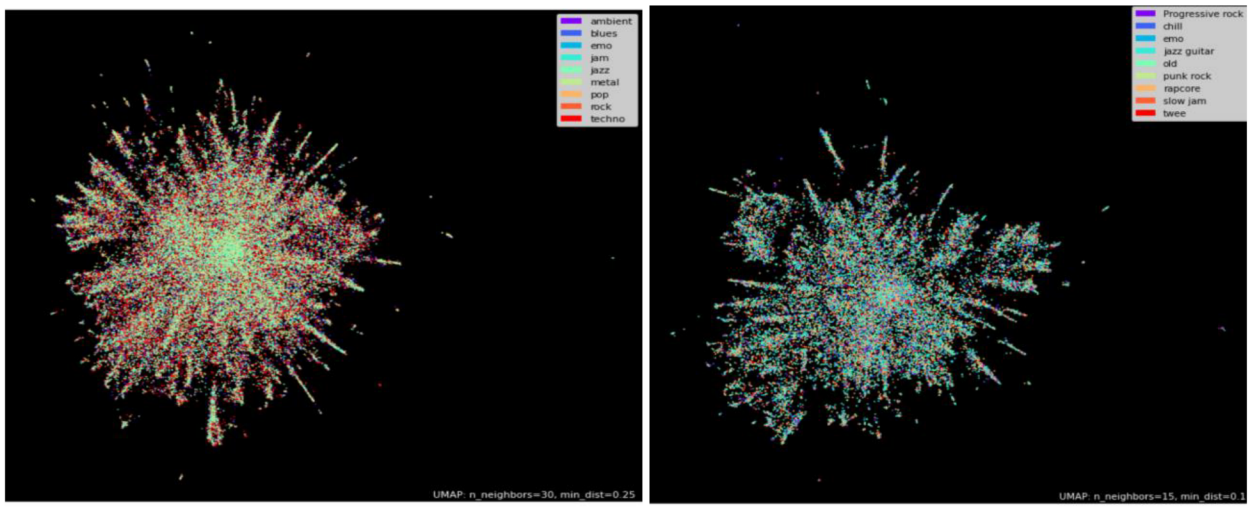
We implemented a baseline popularity method based on the number of user interactions with each track. We aggregated the training data to identify the top 500 songs listened to by the highest number of distinct users, we then added this list to each row of the validation and test datasets. Thus, we recommended this list to all users, and calculated MAP based off of this recommendation set. The resultant MAP value was significantly lower than the baseline ALS implementation. Additionally, as a proof of concept we implemented the LensKit bias module which uses Bayesian dampening to regress extreme values towards a mean and we see that for the same subsample size, we get a much lower nDCG score.

Subsample	**Algorithm**	**Results (vs test)**
1% training subsample	LensKit bias, single machine	nDCG = 0.0004096331928911953 1063.0135743618011 seconds
100% train + validation (popularity model)	User interaction count	MAP: 0.0025827659917961386
100% train + test (popularity model)	User interaction count	MAP: 0.002630705238415203

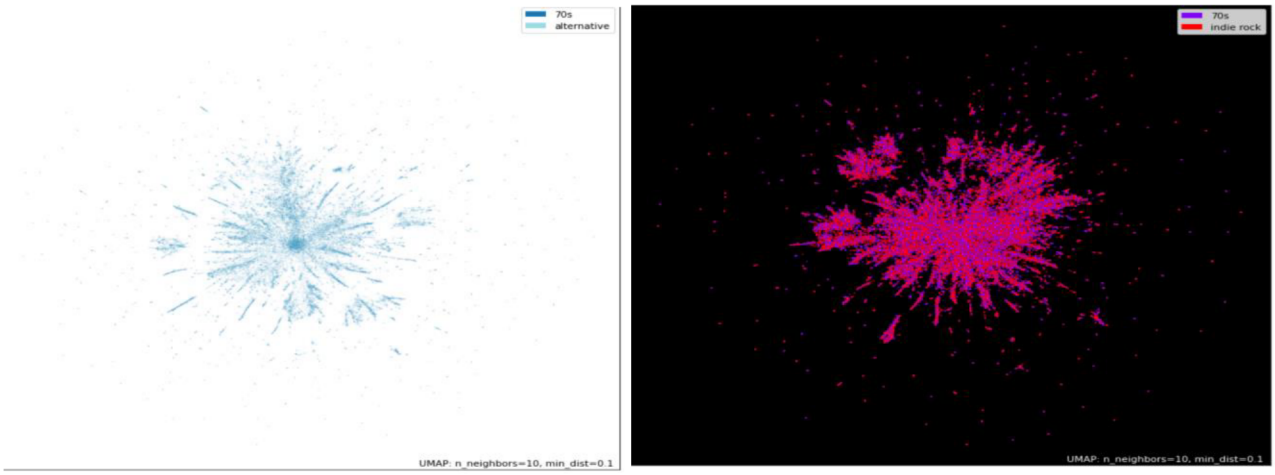
[1] application_1619268950684_19247
[2] application_1619268950684_19539
[3] application_1619268950684_19597

3.3 Extension 3: Item Factor Matrix Visualization using UMAP

We used the 'lastfm_tags.db' to extract metadata describing the tracks. The goal was to represent the item factor matrix of dimensions = rank of 205 in a 2D space using methods like PCA/TSNE/UMAP and identify clusters of latent representations represented by the genre of tracks. We explored two methods - (1) Projecting the latent representations into 10 dimensions using PCA and then using T-SNE to further reduce them to 2. (2) Projecting the latent representations directly into 2 dimensions using UMAP. UMAP was considerably faster than T-SNE. Since there were approximately 500,000 distinct genres, we only considered the ones which had at least 10,000 unique tracks associated with them. We further reduced these genres by grouping them into broader genres detailed in our 'broad_tags' csv. We then plotted the 10 most popular genres to obtain the first visualization and the second plot is for genres that ranked between 10 and 20.



We tried multiple ways of clustering this data - for tracks with multiple genres, we mapped them to both the most popular genre and the least popular one (so that we get the genre that makes the track most unique). We also plotted one vs rest plots for some popular genres to see if we could locate a specific cluster for that genre in comparison to the rest. We also chose genres that could be considered complete opposites like 'country vs house' or 'jazz vs metal' to see their spatial distribution.

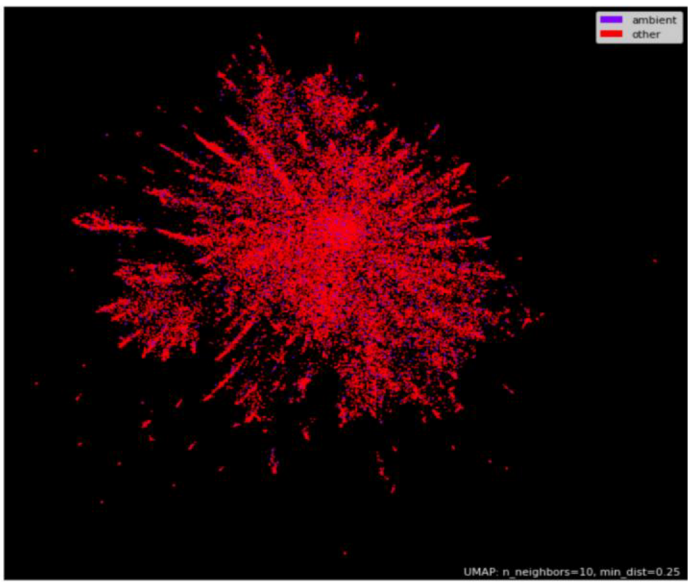


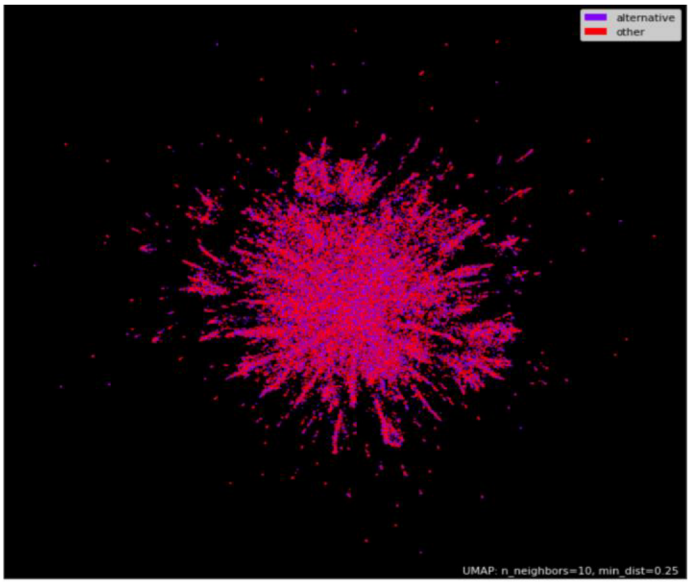
A. Appendix

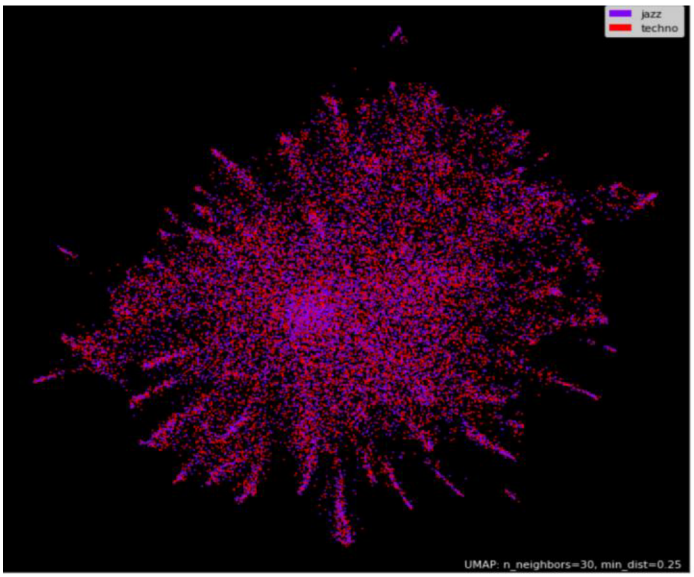
Appendix A-1: Sample of hyperparameter tuning results

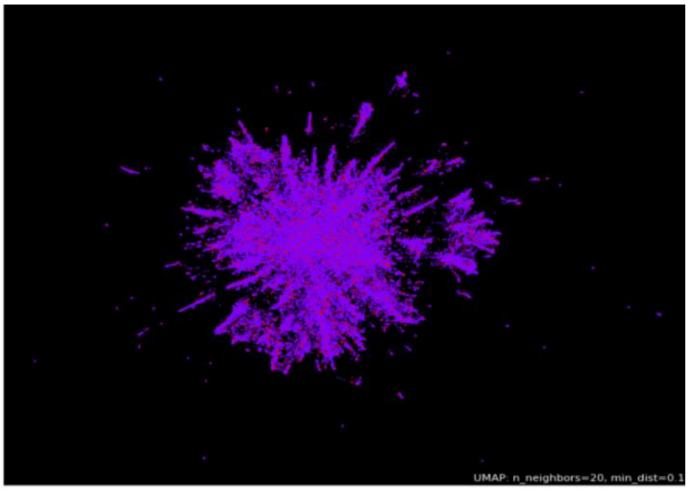
Rank	Regularization	Alpha	MAP (validation)	Application ID
5	0.01	1	0.0333073818469817	application_1619268950684_16621
5	0.1	1	0.0333073818469817	application_1619268950684_16621
5	0.2	1	0.0333073818469816	application_1619268950684_16621
205	0.01	100	0.0791694714922161	application_1619268950684_17146
210	0.01	10	0.0695188566631704	application_1619268950684_16583
210	0.01	20	0.0733364423728588	application_1619268950684_16583
210	0.01	50	0.077453244267661	application_1619268950684_16583
250	0.01	100	0.0722184567951454	application_1619268950684_17146
300	0.01	100	0.0733162938947042	application_1619268950684_17146
310	0.01	100	0.0740919747056026	application_1619268950684_17146

Appendix A-2: Extension 3 additional visualizations









B. Contributions

- Pavel Gladkevich: Data Preprocessing, Hyperparameter Tuning, Extension 2 (Popularity / Bias model), Write-up
- Surabhi Ranjan: Data Preprocessing, Baseline Model, Hyperparameter Tuning, Extension 2 (Lenskit bias execution on Greene), Extension 3 (Visualization using UMAP), Write-up
- Ted Xie: Baseline Model, Hyperparameter Tuning, Extension 1 (Single machine implementation), Extension 3 (PCA/TSNE) , Write-up
- Duey Xu: Baseline Model, Hyperparameter Tuning, Extension 1 (Single machine implementation), Write-up