Music Recommendation Systems on Million Song Dataset (MSD)

New York University Center for Data Science Spring 2021 DSGA-1004 Big Data Final Project

Claire Ellison-Chen
te2049@nyu.edu

Yi Wen
yw5280@nyu.edu

Zhuoyuan Xu
zx1137@nyu.edu

1 Introduction

The project aims to create and evaluate a collaborative filtering-based song recommender system with Alternative Least Squares (ALS) model in Apache Spark via Python, using the interactive records of users and their listened tracks from the Million Song Dataset stored on Hadoop Distributed File System (HDFS). The resulting ALS model was then compared with single-machine LightFM implementation on their relative efficiency and accuracy. Finally, the learned representation of tracks from our model was visualized in UMAP.

2 Basic Recommendation System (ALS)

2.1 Data Overview

The training set derived from the Million Song Dataset contains interaction histories of 1,129,318 distinct users and their corresponding listening activities, with a validation set of 10,000 users and a test set of 100,000 users. The dataset takes in the form of implicit feedback, with each row composed of a user ID (string), a track ID (string) and the corresponding play count (positive integer). The training set includes partial histories of users in the test and validation sets, and the complete records for the rest of the users.

2.2 Data Processing

Feature Transformation. To meet the format requirements of ALS, the StringIndexer function was applied to the training set to convert its user IDs and track IDs from string to numeric indexes. The two separate indexers were combined into a pipeline to transform user IDs and track IDs in the validation and test sets and skip unseen labels via *setHandleInvalid("skip")*.

Downsampling. Due to the sheer size of the dataset, a downsampling method on the training set was considered in regard to the computational cost in time and memory. To ensure that the downsampled data include enough users from the validation and test set for evaluation, instead of random sampling of entries, we first selected all the interactions of users in the test and validation sets from the training set, sampled a fraction of the rest of the unique user IDs in the training set and combined with these selected users' interactions for model training. The learning curve graph below shows the sampled fraction of user IDs against the root-mean-square error (RMSE) of the ALS model using the default settings at *max_iter* =1. We selected the result at the elbow of 50% sampling for our formal modeling as we consider the RMSE at this point is sufficiently low, different by less than 5% from that of the full set.

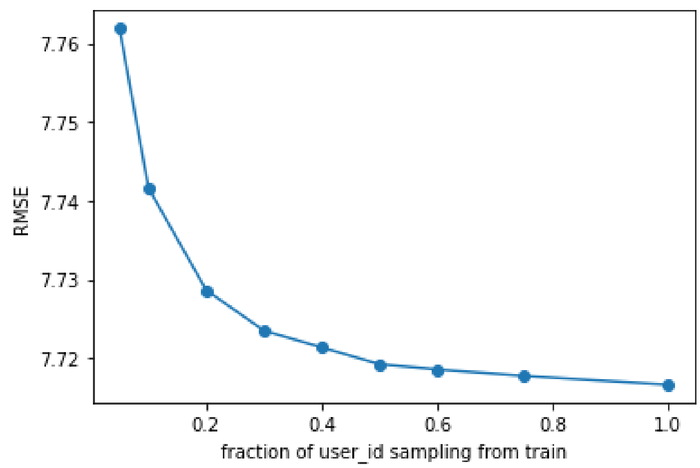

Figure 1: *Learning Curve (RMSE) for Different Fractions of User IDs Sampled from the Training Set*

2.3 Model Implementation

2.3.1 ALS Model

Our recommender system uses collaborative filtering to recommend the top 500 tracks for each user. Alternating Least Square (ALS) is a matrix factorization algorithm that can construct latent factor matrices of users and products and predict the missing entries of the user-item association matrix. Given our dataset, the ALS approach in *spark.ml* allows us to make personalized recommendations by treating the implicit listening counts as the level of confidence in observed user preference and using the learnt latent factors to predict the expected preference of a user for an item [1]. The model was implemented with settings *implicitPrefs* = True and *nonnegative* = True for implicit feedback, *numUserBlocks* = 50 and *numItemBlocks* = 50 for parallel computation.

2.3.2 Evaluation Metrics

We chose two metrics to evaluate the performance of our recommender, since our goal was not only to recommend songs to users but also to list the recommendations of stronger confidence level in top positions. **Precision at k** measures how many of the first k recommended documents are in the set of true relevant documents averaged across all users. **Mean Average Precision (MAP)** is a similar measurement of the overlap between the recommended documents and the true ones, but taking the order of the recommendations into account by weighing the true relevant items predicted early in the recommended list higher [1]. Among the two chosen metrics, MAP is more suitable for song recommendation since the user is less likely to listen to songs behind in the ranked recommendation list, while precision at k tells the inclusion of our recommendation.

The ranking metrics were implemented by *RankingMetrics* from *pyspark.mllib* which takes in the combined RDD of the true track list and the recommendation results using *recommendForUserSubset* of the fitted ALS model for all users in the validation or test set.

2.3.3 Parameter Tuning

We focused on tuning three hyper-parameters *rank*, *regParam* and *alpha* to improve our ALS model evaluated on the validation set. First of all, we narrowed down values for each parameter for the next step of grid search. With other parameters fixed to default, we tried a wide range of values for each parameter to find their trends. See the table below for the tuning ranges and the default values of the tuned parameters.

Parameter	**Default**	**Tuning range**
rank	10	[10,20,30,40,50,75,100,125, 150]
regularization	1	[0.001, 0.005, 0.01, 0.1, 0.2, 0.5, 1, 10]
alpha	1	[0.5,1, 5, 10,20,30,50,80]

Table 1: *Default Values and Tuning Ranges of the Tuned Parameters for ALS*

Representing the number of latent factors in the model, a higher rank indicates more information in the learned factorization matrices and hence returns higher evaluation scores. However, higher rank comes with longer training time and the risk of overfitting. Therefore, we will take the highest rank in our range at 150. In addition, the regularization to control overfitting prefers smaller values with a peak MAP around 0.1. Finally, the alpha is applicable to the implicit feedback that governs the baseline confidence in observed preferences. Our preferred value would be around 10 considering both MAP and Precision scores.

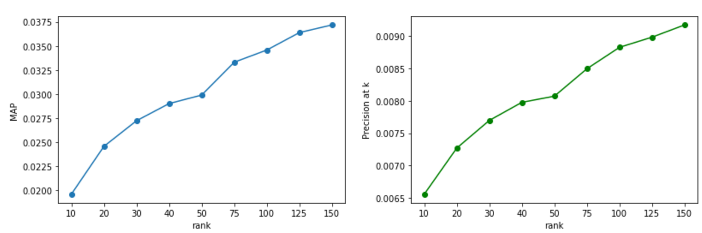

Figure 2: *Spark ALS Performance w.r.t Rank. (Left) Mean Average Precision (MAP). (Right) Precision at K.*

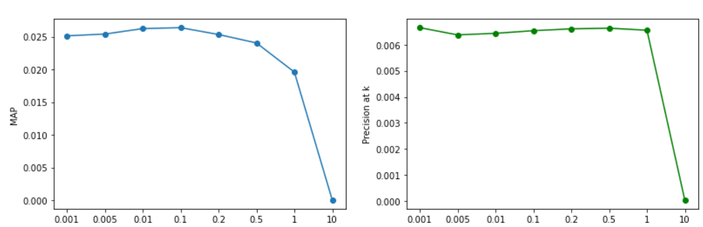

Figure 3: *Spark ALS Performance w.r.t Regularization Parameter. (Left) Mean Average Precision (MAP). (Right) Precision at K.*

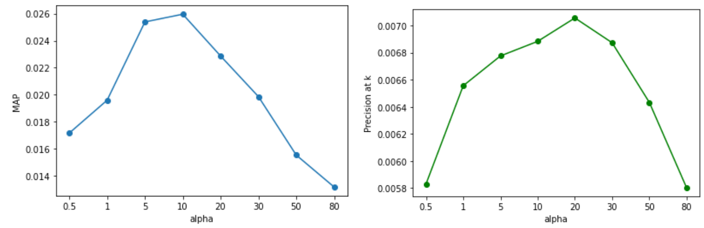

Figure 4: *Spark ALS Performance w.r.t Alpha. (Left) Mean Average Precision (MAP). (Right) Precision at K.*

Then we performed grid search on regParam in [0.05, 0.1, 0.15] and alpha in [7.5, 10, 12.5] at rank = 150 to look for the best combination:

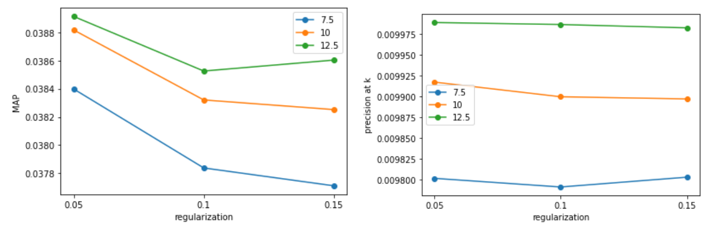

Figure 5: *Spark ALS Performance w.r.t Regularization Parameter for Different Alpha. (Left) Mean Average Precision (MAP). (Right) Precision at K.*

Based on both ranking metrics on the validation set, the hyper-parameters to achieve the best performance are *rank* = 150, *regParam* = 0.05 and *alpha* = 12.5. The final model trained on the downsampled dataset returns the following result on the test set:

MAP	**Precision at k = 500**
0.0387	0.00996

Table 2: *Result of the Best Performing ALS on Test Set*

3 Extension 1: Comparison with Single-Machine Implementation LightFM

After evaluating our basic Spark ALS model, we compared its efficiency and accuracy with respect to training file size with a single-machine implementation, LightFM. LightFM is a hybrid matrix factorization algorithm which incorporates collaborative and content based filtering to make recommendations. It represents both users and items as linear combinations of their content features' latent factors. Meanwhile, it can be built from both explicit and implicit feedback. Particularly, it supports two models that are suitable for implicit feedback: Bayesian Personalized Ranking (BPR) pairwise loss [3] and Weighted Approximate-Rank Pairwise (WARP) loss [4]. In this section, we compared our Spark ALS model with both LightFM BPR and WARP. We fed the models with the same training set, and adjusted the percentages of its size used for fitting, ranging from 0.1% to 30%. We did not include metadata into the LightFM fitting process because we would focus on the training size, the variable of our interest. We also fixed regularization parameter to 0.05, maximum iteration to 1 and rank to 150 in our entire experiment. We demonstrate the differences in model efficiency by fitting times (Figure 6), and recorded the corresponding precision at k as the accuracy metric (Figure 7), consistent with our evaluation for the ALS model in the previous sections.

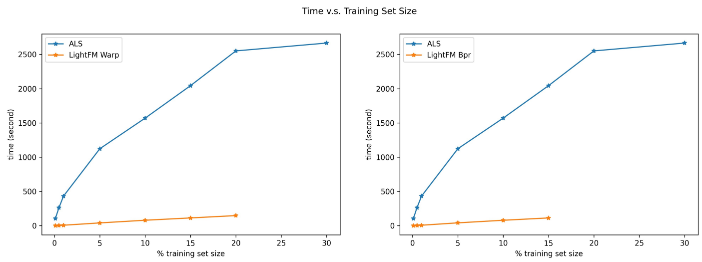

Figure 6: *Fitting Time w.r.t Percentage of Training Set Size. (Left) Spark ALS and LightFM WARP. (Right) Spark ALS and LightFM BPR.*

Figure 6 shows the fitting time in seconds under various percentages of training sizes for our ALS and LightFM models under our setting. We ran the experiment multiple times and the results were consistent. We noticed as the percentages of the training size reached about 20%, both LightFM models led to the dead kernel. On the contrary, Spark ALS was able to run through all chosen sizes. This may result from the design of LightFM as an implementation on a single machine which has limited CPU and other resources. In addition, we adjusted our LightFM parameters and environment setup to validate our hypothesis. As we increased the maximum iteration and the rank, the kernel may die at smaller training sizes, whereas if we increased the machine memory, we could implement LightFM for larger file sizes.

We can see that ALS took longer than both LightFM WARP and BPR. Here, WARP and BPR had similar fitting times, probably because they only differ in their loss functions. Many reasons may account for the slow speed of Spark ALS. For example, since Spark is designed for large-scale, paralleled data processing on distributed systems, it could be slow if too many tasks run concurrently [5] with abundant communication among nodes required. Also, Spark was originally written in Scala and ran on Java Virtual Machine so running Spark on Python could slow down the development process due to the data transformation between Python interpreter and virtual machine [6]. It is also worth noting that the increase rate of ALS fitting time decreases as file sizes become larger. This finding indicates potential benefits of ALS for processing large files.

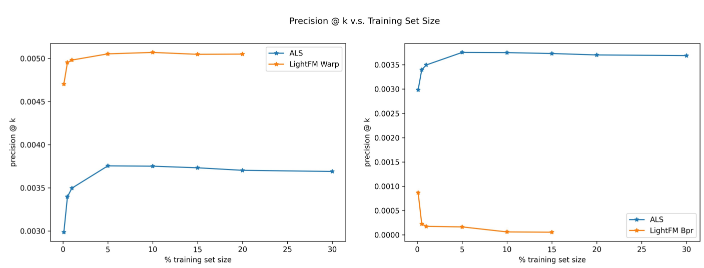

Figure 7: *Precision at K w.r.t Percentage of Training Set Size. (Left) Spark ALS and LightFM WARP. (Right) Spark ALS and LightFM BPR.*

The precision at k values with respect to the training set size is shown in Figure 7. Both LightFM WARP and Spark ALS values increase initially until reaching a plateau. Such a behavior is expected because model performance increases dramatically in the beginning as more training data gets incorporated into training until data saturates and any additional data would not improve the learning. The higher precision at k of LightFM WARP is consistent with the original research paper which claimed that LightFM generally outperforms both collaborative and content-based models in cold-start or sparse interaction scenarios with user and item metadata, and performs at least as well as a pure collaborative matrix factorization model with abundant interaction data [2]. The choice of loss function could also lead to higher precision at k. WARP aims to optimize the top of the recommendation list [7], while Spark ALS uses more general alternating least-square function. On the other hand, LightFM BPR produces decreasing precision at k. According to the LightFM documentation, BPR maximizes the prediction difference between a positive example and a randomly chosen negative example, and it is useful only when positive interactions are present and optimizing ROC AUC is desired [7]. Therefore, BPR may not be suitable for our specific dataset, and the metrics of our interest may not align with the goal of the model.

In summary, our comparison provides basic, qualitative observations of different behaviors of Spark ALS and LightFM models with our chosen setting. LightFM is potentially more efficient for smaller data sizes, and can give higher accuracy depending on the dataset and the overall goal of the recommender. Spark ALS may be more powerful for large-scale data with paralleled tasking, and it may show more advantages as data scale increases. Meanwhile, LightFM is able to incorporate content-based information and perform hybrid recommendations. The two algorithms benefit different types of development, and their performances vary under project needs.

4 Extension 2: Exploration with UMAP

The latent factors learned from the ALS model for both items and users (itemFactors, userFactors) are high-dimensional data and can be visualized using UMAP [9]. UMAP (Uniform Manifold Approximation and Projection) is a novel manifold learning technique for dimension reduction and is constructed from a theoretical framework based in Riemannian geometry and algebraic topology [9]. In this project, we visualized the item latent factors against the track genres. We dropped the user factors since we lack an appropriate user related target for visualization.

4.0.1 Data Preparation for UMAP

Preparing the data for UMAP visualization, we extracted the item factors learned by both ALS models (default and the best performing) and joined them with the music track metadata to include the track genres as our target label for visualization. The track genre tag is cleaned from the metadata following the steps below:

1. For each track, we got a list of genres with the highest associated scores. e.g. rock: 100, pop: 100, american: 50 outputs [rock, pop]. The tracks that lack genre information are labeled 'NA'.

2. Then for each track, we got the most popular genre tag from the list of highest scored genres. e.g. [rock, pop] outputs 'rock' because 'rock' is a more popular genre than 'pop'. Fortunately, there aren't any ties in popularities in this step of data cleaning and it results in a single genre tag for each track.

3. Finally, to facilitate interpretable and analyzable visualization, we only labeled the track genres that appeared in the top 13 most popular genres, and label the rest as 'other' genres.

In the above data cleaning processes, we defined the popularity of a genre tag as the number of times the genre appears in the entire track genre space. Note that, in this way, each track will be associated with one most dominantly popular genre even though a track is defined by multiple, sometimes equally highly scored, genres.

After data cleaning, we had a dataframe with 482,149 tracks each with 150 item latent factors and a single genre tag. The size of the top 15 genres vary from the leading ones like other (297,541), na (72,713), and rock(45,693) to the end of the list like American (259), in a way that a few genres label the majority of tracks. The 'other' genre is dominant due to the sparsity of the genre space and 'NA' follows due to the lack of genre metadata of many tracks.

4.0.2 Parameter Tuning for the best UMAP visualization

Guided by the tuning process for our ALS model, we downsampled the visualization data to roughly 20,000 rows for faster tuning time and focused on the following parameters in the ranges in Table 3.

Parameter	Default	Tuning range
n_neighbors	15	[20, 50, 100, 200, 300, 400, 500, 1000, 1500, 2000]
min_dist	1	[0.1, 0.5, 0.75, 1, 1.5, 2, 3, 5, 10, 50]
alpha	0.1	[1e-8,1e-7, 1e-5, 1e-4, 1e-3, 0.01, 0.05, 0.1, 0.5, 1]

Table 3: *Default Values and Tuning Ranges of the Tuned Parameters for UMAP*

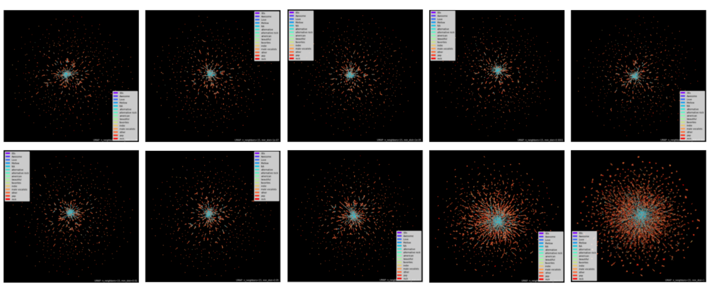

Figure 8: *Size of Clusters Increases w.r.t min_dist Increase*

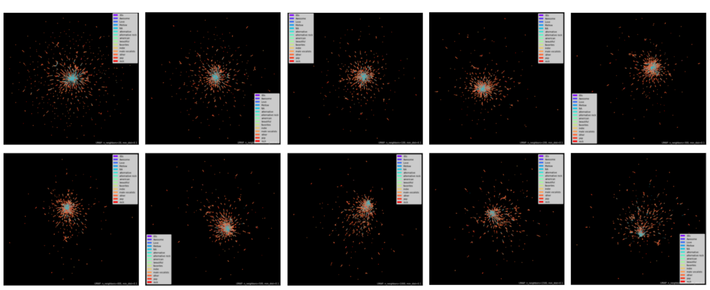

Figure 9: *Size of Clusters Collapses w.r.t n_neighbors Increase*

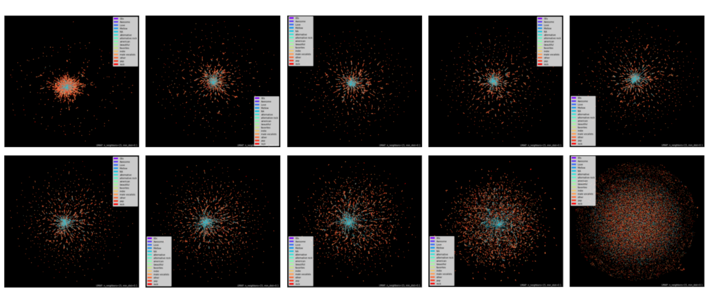

Figure 10: *Size and Boundary of Clusters Increases w.r.t spread Increase)*

After tuning, we found that *n_neighbors* doesn't change the general UMAP shape much (see Figure 8). The larger *n_neighbors*, the more concentrated each cluster of points is, whereas the change is subtle. The shape doesn't change much after *n_neighbor* = 300 but mapping time increases dramatically. In addition, both *spread* and *min_dist* could impact the general shape (see Figure 9, 10). As *spread* or *min_dist* gets larger, the clusters grows closer with less boundary (less differentiation) between them. The run time, however, doesn't change much when we increase *spread* or *min_dist*. The final parameter ranges for further grid search are shown in Table 4.

Parameter	Tuning range
n_neighbors	[100, 200, 300]
min_dist	[0.5, 0.75, 1]
alpha	[0.05, 0.1, 0.5]

Table 4: *Default Values and Tuning Ranges of the Tuned Parameters*

Eventually, we plotted our final visualization of the item latent factors using *n_neighbors* = 200, *min_dist* = 0.5, *spread* = 0.75.

4.0.3 Result and Analysis

The visualization of the outputs by the default ALS model (left) and our best performing ALS model (right) against the track genre tags are displayed below in Figure 11.

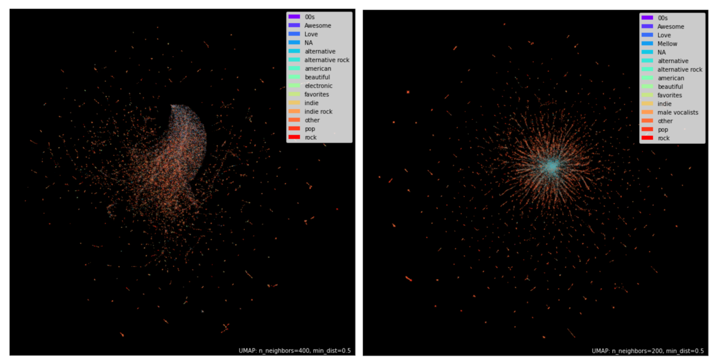

Figure 11: *Latent Itemfactors from Default ALS Model (Left) and the Best Performing ALS Model (Right) visualized against the Track Genre Tags*

The graph shows that our best performing ALS model beats the default ALS in terms of differentiating tracks with a genre from those without a genre. For the default ALS model, all tracks of different genres are mixed together while the tracks of the 'NA' genre (blue dots) just start to show a hint of separation from the rest. On the contrary, there is a clear boundary between the tracks without a genre (blue dots) from the most popular genres (red dots) for our final model. The trend that the item latent factors of ALS learned to differentiate tracks without a genre from the rest may make sense in real life, because the tracks lacking genre metadata are less likely be recommended to users, resulting in a plausible user behavior that get implicitly learned by ALS.

Despite differentiating tracks without genre from those with one, the learned item factors implied little on the distinction among track genres. Genres 'rock', 'pop' and 'other' scatter all over the learned space, possibly due to the limitation that one track could have been associated with multiple equally important genres, whereas we only selected one defining genre and allowed popular genres such as 'rock', 'pop' to dominate the genre space. Hence, those tracks labeled 'rock' and 'pop' might differ in reality with other classifications, which are learned by our best ALS model, causing these tracks scatter across the whole space instead of forming clusters.

References

[1] "Collaborative Filtering - Spark 2.2.0 Documentation." Spark, spark.apache.org/docs/2.2.0/ml-collaborative-filtering.html. Accessed 18 May 2021.

[2] "Collaborative Filtering - Spark 2.2.0 Documentation." Spark, spark.apache.org/docs/2.2.0/ml-collaborative-filtering.html. Accessed 18 May 2021.

[3] Kula, Maciej. "Metadata Embeddings for User and Item Cold-start Recommendations." ArXiv abs/1507.08439 (2015): n. pag.

[4] Rendle, Steffen, et al. "BPR: Bayesian personalized ranking from implicit feedback." Proceedings of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence. AUAI Press, 2009.

[5] Weston, Jason, Samy Bengio, and Nicolas Usunier. "Wsabie: Scaling up to large vocabulary image annotation." IJCAI. Vol. 11. 2011.

[6] "Why Is Spark So Slow? (How Can I Fix Things?)." Pepperdata, 5 Jan. 2021, www.pepperdata.com/blog/why-is-spark-so-slow.

[7] Bennett, Robert. "The Good, Bad and Ugly: Apache Spark for Data Science Work." The New Stack, 26 June 2018, thenewstack.io/the-good-bad-and-ugly-apache-spark-for-data-science-work.

[8] "An Implicit Feedback Recommender for the Movielens Dataset — LightFM 1.15 Documentation." LightTFM, making.lyst.com/lightfm/docs/examples/movielens_implicit.html. Accessed 18 May 2021.

[9] Leland McInnes, John Healy, James Melville, UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction, arXiv:1802.3426

A Appendix: Contributions

Member name	**Contribution**
Claire Ellison-Chen	Basic recommender system (ALS)
Zhuoyuan Xu	Parameter tuning, extension1 (single machine)
Yi Wen	Metadata cleaning, parameter tuning, extension2 (UMAP visualization)

B Appendix: Files and functions

Basic Recommender (ALS):
sample_indexer.py: downsample and StringIndex
learning_curve.py: learning curve to determine training set size
param_train_1st.py, param_train_2nd.py: parameter tuning for each parameter and for grid search
one_train.py: training a single model with set parameters

Single-Machine Implementation (LightFM):
als_model_extension_2.py: Spark ALS (rank=150, regParam=0.05, max_iter=1)
extension2_1004project.ipynb: LightFM implementation

Exploration (UMAP):
Exploration-EDA.ipynb: outputs cleaned data df_final.csv
UMAP visualization.ipynb: UMAP tuning and visualization
Genre_10.csv: top 13 most popular track genres tags
df_final.csv: music metadata for UMAP visualization
dominant_trackgenre.csv: track genre by index order of df_final.csv
Item_matrix_full.csv: best ALS model learned item factors (on 5% training data)
track_ids.csv: ALS track string index versus track_ids; used to join metadata with item factors