1004 Yuge Data Group Final Project

In this report, we used the Million Song Dataset to build a recommender system for users. The Million Song Dataset itself contains one million songs, along with corresponding features such as metadata and tags, among others. We were also provided with a dataset containing user-song interaction data for one million users on these one million songs, which measures interactions by play count for each user-song pair.

The computational problem we addressed was to use this dataset to build a recommender system for the users. In particular, we wanted to predict 500 ranked recommended songs for each user, given each user's user-song interaction data. The user-song interaction data can be represented as a utility matrix containing interaction data for user *i* on song *j* at row *i,* column *j* of the matrix. The task at hand was thus to predict the missing entries of the utility matrix. To do this, we used a latent representation with matrix factorization approach. Instead of directly predicting the missing entries, we tried to predict two matrices, a user matrix and an item matrix, whose product was the full utility matrix. These two matrices had respective dimensions of *n* x *f* and *f* x *m*, where *n* was the number of users, *m* was the number of songs, and *f* was the number of latent factors, which was a hyperparameter we defined and tuned ourselves. These latent factors were a means for us to represent hidden features pertaining to how users and songs interacted in a very low dimensional space, and we could use them to predict future user-song interactions. The higher the number of latent factors, the more improved our model would perform on our training set. However, this also came with a higher risk of overfitting, so we also incorporated an L2 regularization parameter.

To obtain our desired matrix factorization, we fit our dataset using Spark's Alternating Least Square method. The ALS method is extremely useful because it allows us to perform our matrix factorization task in parallel. The method aims to minimize the norm of the difference between the true utility matrix (using entries given in our data) and the predicted utility matrix while respecting the regularization constraints. It does so by alternating between minimizing two separate loss functions corresponding to the user and item matrices, holding one constant while taking one gradient descent step on the other.

For the implementation of this project, we first used a Spark StringIndexer on the users and the tracks to make them uniform and faster to compute on. We then fit a Spark CrossValidator with an ALS model and an RMSE (root mean squared error) evaluator with 5 folds to determine the best hyperparameters. We use the root mean squared error as opposed to regular mean squared error because the root mean squared error puts more weight on larger errors. We fit the best model from the CrossValidator on the training data and evaluated the RMSE of the transformed test data. We also turned the test set results into a csv file for further analysis in the extensions, particularly the UMAP for visualization, and comparison with the baseline popularity model.

We had initially tried running each hyperparameter in a for loop, but with the long runtimes of large rank sizes we found that it was more computationally efficient with Spark' s CrossValidator class that has in it a parallelism option. This allowed multiple hyperparameter searches to be performed on the cluster simultaneously, and saved many hours of time.

We tuned hyperparameters over the following ranges on 1% of the dataset with 5-Fold cross validation using the spark CrossValidator class to search in parallel natively using spark. The number of ranks tested were 5, 10, 15, 50, 100, 150, 200, 225, 250, 275, 300, and 400. The regularization parameters were 0.001, 0.01, 0.1, 1.0, 10.0, and 100.0. We also attempted to tune the alpha parameter over the range of 1.0,   10.0, and 100.0, but changing it seemed to have little effect on the RMSE, and made the grid too large, due to more combinations, to effectively search on the cluster. This hyperparameter search gave the best RMSE on the validation set with the optimal parameters of 300 ranks and a regularization parameter of 1.0. The RMSE for this setup was 7.7639 on the validation set and 7.1603 on the test set.

Extension 1: Popularity-baseline model

There is a systematic tendency that some users give higher ratings than others, and the same exists for some items receiving higher ratings than others. The collaborative filtering model captures user-item interactions, and the signal that is seen between them. Thus, it is important to make sure our predictors are accurate by concentrating solely on the relevant signals that represent the true user-item interaction. Thus, this is why having a baseline model is important to compare against.

For our popularity-baseline model we downsampled using 1% of the data as our training set, which we then used to group by track_id, along with the average of song counts. The mean of the count data was used as the prediction, and this was joined with the validation and test sets on the same track_id. Finally we compared each of these joined spark dataframes with the real counts using the RMSE evaluator. The RMSE on the validation set was 7.454, and the RMSE on the test set was 7.685. Compared to our model with tuned hyperparameters, the popularity-baseline model's validation RMSE did much better ($7.454 < 7.7639$). However, the RMSE on the test set with the tuned hyperparameter model is much lower than the popularity-baseline model's RMSE ($7.1603 < 7.685$). This means that the tuned hyperparameter model with parameters of 300 ranks and a regularization term of 1.0 generalized better on the out-of-sample data.

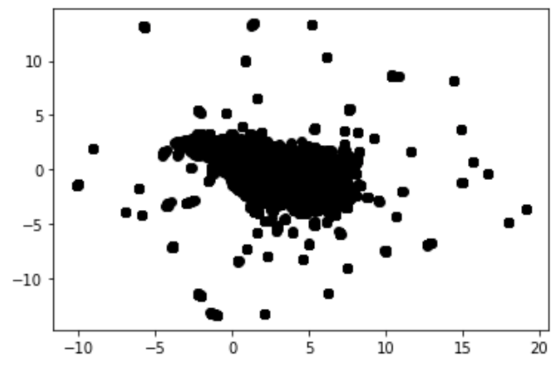
Extension 2: UMAP Visualization

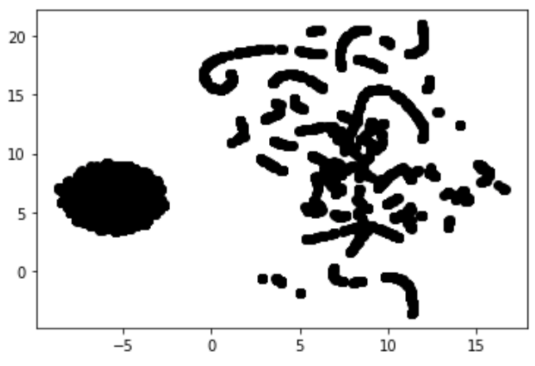
In order to perform visualization on the model output, we used the UMAP algorithm to find a low-dimensional embedding of our data that preserves the original structure. Using a more limited sample from our model output, we can map the user_id, track_id, count, and ranking features into a low-dimensional representation, which is plotted using the seaborn package.

Since we applied the UMAP algorithm to a smaller dataset, we ran the code locally to achieve the visualization. First, we convert the dataframe of the top 500 song recommendations into a csv file. We then read the csv locally using pandas, and fit the data using the fit_transform function from the UMAP package. Using the 'spectral' cmap parameter yields a spectral embedding of the fuzzy 1-skeleton - or a topological graph of the data.

In the 2-dimensional space, we can observe how various clusters are represented using colors, such that points that were nearby in the higher dimensional space are closer to each other in the 2D graph. From this output, we can also adjust the parameters to control the UMAP's clustering of the data. For example, the n_neighbors parameter places a size constraint on the neighborhood that UMAP looks at to determine the manifold structure. As n_neighbors increases, UMAP captures more of the overall structure.

After concatenating the csv files from the recommendation output, we manipulate the data by assigning each user to one row, and each column to one track. We then fit the data using UMAP and plot the first two columns of the resulting dataframe, which was reduced by UMAP.





The left plot represents the clusters formed by using the entire dataset. For the plot on the right, we also observe various clusters by limiting the dataframe to only the first 10 columns.

Team member contributions:

Amir Malek: Worked on ALS model and UMAP extension. Also contributed to project write up.

Anthony Chen: Worked on ALS model, hyperparameter set up, model's output formatting, wrote up popularity-baseline model

Daniel Tang: Worked on ALS model (debugging), computational problem and background of writeup as well as contributions to other sections

David May: Worked on ALS model. Ran and debugged hyperparameter tuning on the cluster. Worked on implementation of the baseline model. Wrote up implementation and hyperparameter tuning sections.