MSML 651 Final Project Report

Music Recommendation System with Million Song Dataset

Mengxi Shen

Abstract

Systems that can process and understand user's musical tastes and can thus provide song recommendations hold great value to music streaming applications, producers, and consumers. Companies like Pandora, Spotify, and Apple are interested in using production sized songs and user data to understand what types of music and listeners belong together. Music audiences have an abundance of different songs they could choose from, and with the quantity and quality of data available on user music streaming behavior and music content, a better recommendation system is possible.

I. Introduction

A. Problem Statement

In this project, I propose and implement a machine learning pipeline that combines content-based and collaborative recommendation methods for a large-scale, personalized song recommendation system. The goal is to predict which songs that a user will listen to and make a recommendation list of 10 songs to each user, given both the user's listening history and full information (including meta-data and audio feature analysis) for all songs.

To solve this problem, I pose several specific questions to guide my exploration: What requirements need to be satisfied to achieve good performance on collaborative-based recommendation? Which collaborative-based recommendation algorithm is suitable for solving production-sized problems? How to measure similarities between songs? How to design a suitable content-based recommendation method for this dataset? How to measure recommender system performance?

B. Datasets

Two datasets are involved in this project. The first one is Million Song Dataset (MSD), a freely-available collection of extensive meta-data, audio features, tags on the artist and song level, lyrics, cover songs, similar artists, and similar songs for a million contemporary popular music tracks. The raw dataset is 280GB and is available on AWS as a public dataset snapshot. The dataset identifies a song by either song_id or track_id. One or the other is consistently used in the complementary datasets, which allows me to correctly preprocess the data. Given one of the id's I can then determine the song name and the artist. Due to copyright issues, the MSD does not provide audio samples of songs but provides derived audio features such as chroma and Mel-Frequency Cepstral Coefficients (MFCC) features. The derived audio features are time-series data of up to 945 timesteps for a given song. Each song has a different number of timesteps and the majority of the songs had timesteps in the 300 and 400 range. In addition, timesteps can have different time length within a song or between songs. I observe that the maximum time of a timestamp is 5 second, while 99% of the time lengths are less than or equal to 1 second. In addition, MSD provides timbre information, an important feature for musical information. Timbre describes the perceived sound quality of a musical note, sound, or tone. Timbre distinguishes different types of sound production, such as human voices and musical instruments like string instruments, wind instruments, and percussion instruments. The timbre features at every timestep are computed by retrieving the MFCC and then taking the top 12 most representative components. The second one is Taste Profile Subset, a dataset consists of userID, songID, and playCount curated by EchoNest consisting of 1,019,318 unique users, 384,546 unique MSD songs, and 48,878,586 triplets, see schema in Figure 1.

user	song	playCount
0007c0e74728ca9ef...	SODMFNK12AF72A2A27	1.0
0007c0e74728ca9ef...	SOTSXKF12A6701C3AF	2.0
0007c0e74728ca9ef...	SOXUQFV12A6310D8AC	1.0
0007c0e74728ca9ef...	SOZNBQP12A6310D8AA	1.0
000b474f815bcff17...	SOPXNZK12A8C13B49D	4.0
000b474f815bcff17...	SOSMMZJ12A6D4FBF22	1.0
000ebc858861aca26...	SOABBNQ12A8C13EFC4	1.0
000ebc858861aca26...	SOAERZE12AB0181FDB	1.0
000ebc858861aca26...	SOAJZLL12A8C1409E7	1.0
000ebc858861aca26...	SOATILC12AB017EC4A	1.0

Figure 1 : Example of User Taste Profile

II. Methods

Two types of methods that are commonly used in recommendation systems are collaborative filtering and content-based filtering. I explored both methods in order to take advantage of the similarities between songs and between users. In this system, preprocessed data are sent to two models, one is collaborative filtering and the other one is content-based filtering. The first model is only used on the Taste Profile dataset, while the second one is used on both MSD and Taste Profile. The recommendations from two models are then combined and ranked to gain the final results. Figure 2 shows the pipeline of the recommendation system.

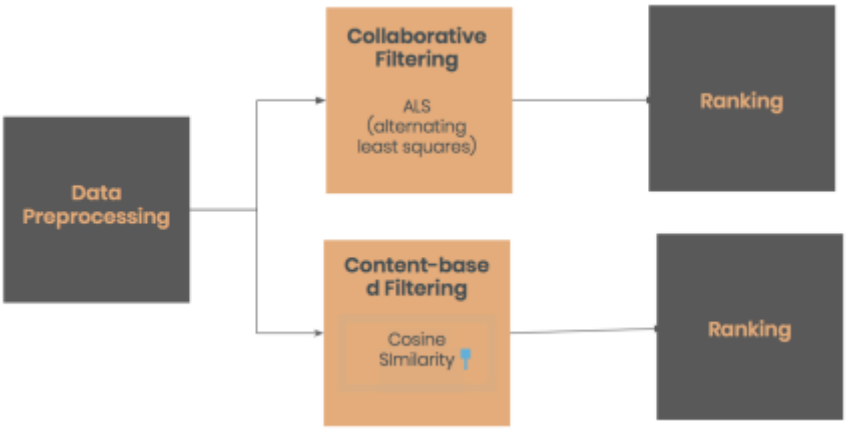

Figure 2: Pipeline of the System

A. Collaborative Filtering

Collaborative filtering assumes that users that have similar opinions and have made similar decisions are likely to make similar decisions in the future. In collaborative filtering, the system only needs the previous behaviors of users, not the details about the choices.

Matrix factorization is a state-of-art solution for sparse data. In the Taste Profile dataset, the problem of sparse data with respect to collaborative filtering occured because there are 1 million of songs. It is impossible for users to have listening records on all songs. If I spread the records to a user by songs matrix, most of the cells are going to be 0, indicating that the user has never listened to the song. Therefore, I choose matrix factorization to confront the problem of sparse data. For collaborative filtering, matrix factorization is used to decompose the user-song count-of-play matrix into the product of two lower dimensionality rectangular matrices. The product sacrifices accuracy in the process of reducing the dimensionality.

The specific algorithm I use in Matrix Factorization is called Alternating Least Square (ALS). The ALS method works by decomposing the user-item interaction matrix into the product of two lower dimensionality matrices: One matrix can be seen as the user matrix where rows represent users and columns are latent factors. The other matrix is the item matrix where rows are latent factors and columns represent items. The ALS algorithm should uncover the latent factors that explain the observed user to item ratings and tries to find optimal factor weights to minimize the least squares between predicted and actual ratings. The high-level idea of ALS is to alternatively minimize the error of plays (reconstruction error) and L2 regularization. ALS runs gradient descent in parallel, and is available in the PySpark package (pyspark.ml.recommendation).

I load the (user, song, playCount) triplets on clusters. Since I need to put them on a matrix, I choose StringIndexer to convert the user ids and song ids to Integer. Then I use cross validation to select the best hyperparameters (rank and regularization) with RMSE. The final model is saved on AWS S3.

B. Content-Based Filtering

Content-based filtering, on the other hand, builds a model based on the characteristics of an item in order to recommend items with similar properties. From the Million Song dataset, I gained features for each song. Combined with the Taste Profile dataset, the model can leverage the previously listened songs and decide what other songs are similar to the songs that the users have listened to previously.

One common way to measure the similarity between different songs is by calculating the cosine similarity. Given a song, it naturally provides a way to rank other songs based on the similarity scores. For example, for a user, I can recommend songs by ranking their cosine similarities between the user's previously listened songs. In the Million Song dataset, I wish to perform unsupervised ranking of the songs based on selected features, including meta-data and audio features.

However, due to its nature, cosine similarity's performance highly depends on the feature vector. I decided to use cosine similarity with adaptive features weights. That is, when computing cosine similarity, each feature dimension weights differently. Namely, for any two songs song1 and song2, the cosine similarity is:

$$cos(song_1, song_2) = \frac{W \odot x_1 \cdot W \odot x_2}{||W \odot x_1||||W \odot x_2||}$$

where W is the feature weight vector, the operator between W and x implies an element-wise multiplication, x1 and x2 are the pre-processed feature vectors of song1 and song2 respectively. W is trained by maximizing the total cosine similarities of users' previously listened songs.

Pyspark.ml.linalg package makes it possible to calculate this cosine similarity effectively and in a large-scale.

In terms of computation and communication complexity, ALS can partition the matrix computation onto workers and compute updates locally. However, it involves broadcasting the results which results in heavy communication, and matrix computation has high complexity. However, the Taste Profile dataset is relatively small. It was time-consuming in the step of hyperparameter tuning as it needed to build models and test several times.
On the other hand, cosine similarity can be highly parallelized as the cosine similarity needs to be computed for any pairs of songs. In particular, to recommend songs for one user, I need to compute the cosine similarities between his previously listened song and all the candidate songs. Since cosine similarity can only be computed pairwise, the computational complexity is O(nm) where n is the number of user's previously listened to songs and m is the number of all candidate songs. Since the cosine similarity is pair-wise, there is little communication needed. Since the Million Song dataset is much larger and the computation complexity grows up dramatically when n and m are large, I use 1 master and 5 slaves of m5.xlarge instances on AWS EMR.

C. Metrics

In hyperparameter tuning, I use RMSE (of plays) for ALS. In evaluation, I use a separate test set where users have 30% of their listening history hidden. I use the visible 70% of their listening history to predict the hidden song list and evaluate based on the matching between the hidden list and the recommended list. Metrics I use include Precision at k, MAP and NDCG. Precision at k is a measure of how many of the first k recommended songs are in the ground truth (the hidden list of songs users listen to), averaged across all users, where the order of the recommendations is not taken into account. MAP (Mean Average Precision) is a measure of how many of the recommended songs are in the set of ground truth, where the order of the recommendations is taken into account (i.e. penalty for highly relevant songs is higher). NDCG (Normalized Discounted Cumulative Gain) at k is a measure of how many of

the first k recommended songs are in the set of ground truth averaged across all users. In contrast to precision at k, this metric takes into account the order of the recommendations (songs are assumed to be in order of decreasing relevance).

III. Preprocessing and Computation

A. Data Cleaning

The overall data cleaning pipeline is shown in Figure 5. Briefly speaking, the Million Song Dataset snapshot is attached to my AWS EMR cluster, and the raw data is originally stored in HDF5 format and then loaded into spark dataframe. After manually filtering out useless and duplicate features, the intermediate data is stored in AWS S3 as a data structure called parquet.

In the first place, it is noticed that there is an issue of the mismatch between the Million Song Dataset and the users' taste profile dataset. In particular, Million Song Dataset records songs with track metadata while User's Taster Profile Dataset records with song metadata. These mismatches can be found by locating the songs through song metadata and track metadata, and checking the two found songs for consistency. If the two found songs are not the same, it means that mismatch occurs. Fortunately, the Million Song Dataset website provides a list of mismatched songs and songs appearing in the list get filtered out from my data.

B. Feature Transformation

Million Song Dataset contains data of the following data types: float, int, string, array<float>, array<int>, array<string> and array<array<float>>. In addition, the features can be further divided into two main categories: audio features and non-audio features. In the project, I process different categories in a separate manner:

1. "float" and "int": The audio and non-audio features are both included. To process, I normalize them separately with built-in MinMaxScaler and assemble them into a vector.

2. “string”: To start with, I cleaned the original string data based on suggestions provided on the official dataset website. More specifically, I first did the entity resolution on both “song titles” and “artist names”, and then I performed normalization on all string features. Next, I used a built-in feature hasher to convert the string data into sparse vectors. Representing strings in the sparse vector manner is strictly better than storing them with dense vectors since I can store them with terrifically less memory and the computation (e.g. dot product) is also significantly faster.
3. “array<float>”, “array<int>” and “array<array<float>>”: These features are normally audio features, such as “segments_confidence” and “segments_pitches”. For 1-d arrays, i.e. “array<float>” and “array<int>”, the dimensionality relevant to the song length varies. I perform a subsampling strategy on this dimension, i.e. I divide each feature into 10 folds, and for each fold, I select only the most salient one by taking the maximum. In this way, I convert all 1 × N features into 1 × 10. For 2-d arrays, such subsampling policy is not applicable; instead, I use PCA on the song-length dimension and choose only 10 of them. This strategy works since the song lengths are normally larger than 10 bars. With PCA, I convert all 2-d array features with variable size 12 × N into 12 × 10 and later, I unfold these 2-d arrays into 1-d vectors with length 120.
4. “array<string>”: The features of this data type are “artist mbtags” and “artist terms”. For simplicity concerns, I throw away these 2 features since they are difficult to process with variable lengths and thus cannot fit well with the built-in feature hasher.

C. Challenges

One typical challenge that is encountered is processing array<array<float>> data using PCA since it can be computationally expensive. Performing PCA over the features of one million

songs can consume much time even if computation is done in a distributed manner. The other problem is that PCA consumes more memory than expected. At first m5.xlarge instances, which each have 16GB memory, are used as worker nodes. However, worker nodes run out of memory when processing data with PCA. To overcome the challenge, m5.2xlarge instances are used in place of m5.xlarge. Typically, an m5.2xlarge instance has 32GB memory, which enables me to configure Spark such that the maximum memory of a worker node is 24GB. Additionally, m5.2xlarge provides more power in computing and thus helps shorten the time for processing data.

D. Feature Selection

After performing all preprocessing steps on different data types, next, I drop features that are not so informative. For data of type "string", it is unlikely that different features are correlated since different features are of different names and they should occupy different entries after converting into sparse vectors. For data of type array (1-d or 2-d), they are also unlikely to be correlated because higher dimensionalities implies more likely sparsity. Thus, the only features needed to consider are scalar features of type float and int. I first sampled 10,000 samples randomly from the entire dataset (around 980,000 samples after removing mismatched data). Then, I compute the Pearson correlation over these 17 entries, which is formulated as:

$$r = \frac{\sum(x - m_x)(y - m_y)}{\sqrt{\sum(x - m_x)^2 \sum(y - m_y)^2}}$$

I delete two features further ("year" and "energy") since they are all constant in the subsamples and thus informative. I plotted the heatmap over these features regarding their pairwise correlation coefficient in figure 3.

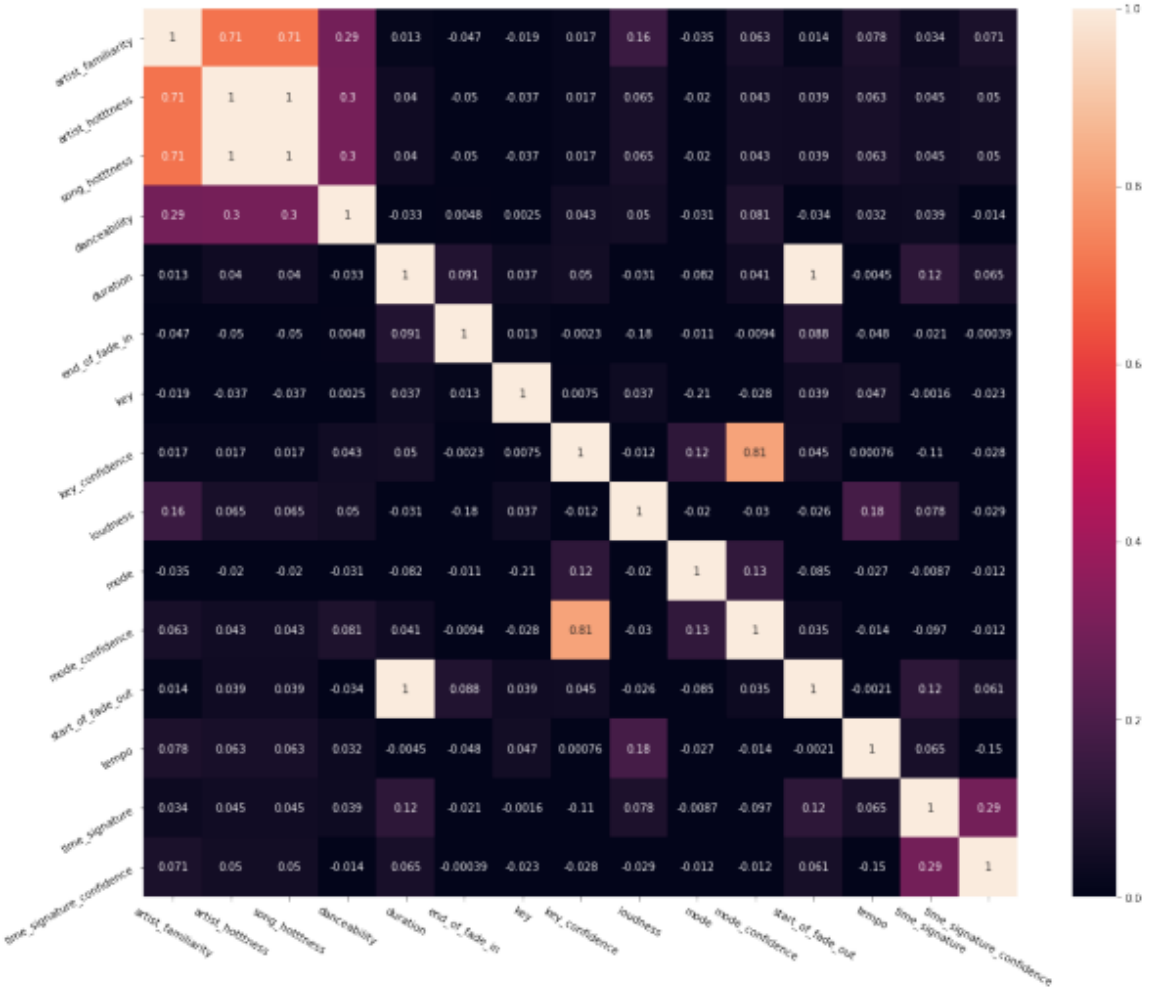

Figure 3: Heatmap over correlation coefficient of pairwise features

By setting the threshold as 0.7, I find these features [’artist familiarity’, ’artist hotness’, ’song hotness’], [’key confidence’, ’mode confidence’] and [’duration’, ’start of fade out’] within each group are highly linear correlated. Thus, I only keep one feature within each group and drop the rest. And finally, 11 / 17 scalar features are selected.

IV. Results

For this project, I devised a straightforward popularity-based baseline algorithm against which to compare my results. The baseline algorithm recommends 10 most popular songs to all users, as shown in Table 1. It has a precision score of 0.0329.

With the ALS method, I first performed grid-search to find the best hyperparameter configuration for ALS. As shown in the result in figure 4, ALS has the best performance when rank = 16 and regularization parameter = 0.25.

```
For rank 4, regularization parameter 0.15 the RMSE is 6.98304734279
For rank 8, regularization parameter 0.15 the RMSE is 6.82760653214
For rank 12, regularization parameter 0.15 the RMSE is 6.71278380904
For rank 16, regularization parameter 0.15 the RMSE is 6.6155780475
For rank 4, regularization parameter 0.2 the RMSE is 6.84832384515
For rank 8, regularization parameter 0.2 the RMSE is 6.67873304069
For rank 12, regularization parameter 0.2 the RMSE is 6.58746190603
For rank 16, regularization parameter 0.2 the RMSE is 6.50708364724
For rank 4, regularization parameter 0.25 the RMSE is 6.76066674977
For rank 8, regularization parameter 0.25 the RMSE is 6.58308072741
For rank 12, regularization parameter 0.25 the RMSE is 6.50156269466
For rank 16, regularization parameter 0.25 the RMSE is 6.4309292434
The best model was trained with regularization parameter 0.25
The best model was trained with rank 16
```

Figure 4: Hyperparameters

Then, using this found hyperparamter configuration, I compare the performance of ALS with the average-playcount baseline on the test set. The RMSE on the ALS is 7.3979, and the RMSE on the average set is 6.1882, while the average number of plays in the dataset is 3.0. I found that the regression metrics RMSE was exceptionally poor for the ALS method. The result is consistent on ranking metrics, as shown in Figure 5 and Figure 6, this ALS configuration does not perform well under the ranking metrics: precision@10, ndcg@10, and MAP. Precision at k is a measure of how many of the first k recommended songs are in the set.

```
%spark.pyspark
from pyspark.mllib.evaluation import RankingMetrics
metrics = RankingMetrics(compare)
print(metrics.precisionAt(10))

print(metrics.ndcgAt(10))

print(metrics.meanAveragePrecision)
```

```
2.0000000000000042e-05
2.1772845250692614e-05
8.25435224594887e-06
```

Figure 5

```
%spark.pyspark

# predict test and rmse
predict = model.transform(dfHidden_idx)
predict = predict.filter(F.col('prediction') != float('nan'))
reg_eval = RegressionEvaluator(predictionCol='prediction', labelCol='playCount', metricName='rmse')
reg_eval.evaluate(predict)
```

```
7.397943655828397
```

Figure 6

In the attempt to arrive at a better result, I hypothesize that by removing inactive users and unpopular songs in their lower quantile would restrict the input data to ALS to have more informative user-item pairs and arrive at better prediction. This step of preprocessing reduces the input data number from the original 48,373,586 to 42,818,160 triplets. After removing inactive users and songs from the input as described above, the ALS achieved better recommendation performance under both the regression metric RMSE and the rank metrics: precision@10, ndcg@10, and MAP, as shown in Figure 10. In particular, the RMSE is 5.0524, which is an improvement from the previous ALS configuration (RMSE = 7.3979), and an improvement from the baseline (RMSE = 6.1882), indicating having informative useritem input is essential for collaborative-based recommendation performance.

```
%spark.pyspark
# print metrics
metrics = RankingMetrics(compare)
print(metrics.precisionAt(10))

print(metrics.ndcgAt(10))

print(metrics.meanAveragePrecision)


# predict test and rmse
predict = model.transform(test)
predict = predict.filter(F.col('prediction') != float('nan'))
reg_eval = RegressionEvaluator(predictionCol='prediction', labelCol='rating', metricName='rmse')
reg_eval.evaluate(predict)
```

```
0.0666666666667
0.0578092189961
0.0238095238095
5.052412445019142
```

Figure 7: Results after Removing

I evaluate my content-based model with recall. Suppose that for one user, I select one song from his or her listening history, and calculate its cosine similarities with all other songs: given a certain threshold, e.g. 0.9, all songs above this threshold are marked as “recommend”, and other songs are not recommended. Suppose the songs lie in the test set are the ground truth data, I could then evaluate the recall for the test set. Note that there is no strictly ”negative” result because the user has no playing history on specific songs does not necessarily mean the user unlike the particular songs. Therefore, I’m only interested in the model's performance on recovering the ground truth (a.k.a recall). The results are in Figure 8. It can be shown that even with a very high threshold (0.9), around half of the ground truth data are recalled, which implies that the model can do a good job on recommendation.

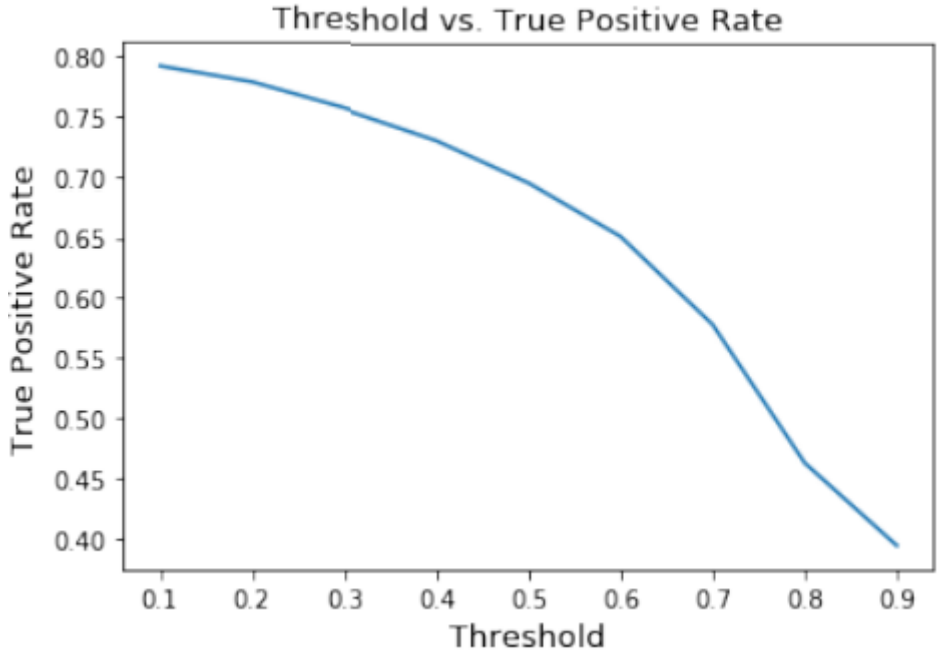

Figure 8

Unfortunately, the results show a generally low performance in terms of precision produced by the song recommendation system. This is expected as most related works revealed a lack of correlation between users’ listening history and their song preferences in the dataset. This also makes me wonder whether precision score is the best evaluation metrics for my problem. Although it is used in the Million Song Dataset Challenge on Kaggle, the best performing

model only achieved a relatively low score of 0.17. Even with a strong model, it's very difficult to predict a set of 10 songs out of 1 million songs that match the users' listening history (the ground truth).