Music to My Ear: Recommender System with Million Song Dataset

Xiaoyi Chen, Zhiran Chen, Kaicheng Ding, Weixin Liu, Xuening Wang, Ruitao Yi
Carnegie Mellon University
{xiaoyich, zhiranc, kaichend, weixinli, xueningw, ruitaoy}@andrew.cmu.edu

Abstract

Systems that can process and understand users' musical taste and provide song recommendations hold great value to music streaming applications, producers, and consumers. Companies like Pandora, Spotify, Apple and more are interested in using production sized song 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 recommender system is possible.

1. Introduction

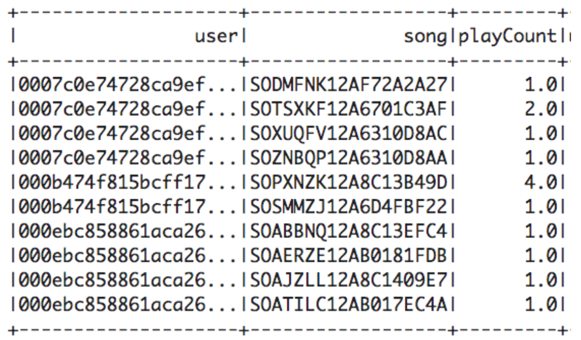
1.1. Problem Statement

In this project, we 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, we pose several specific questions to guide our 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 problem? How to measure similarities between songs? How to design a suitable content-based recommendation method for this dataset? How to measure recommender system performance?

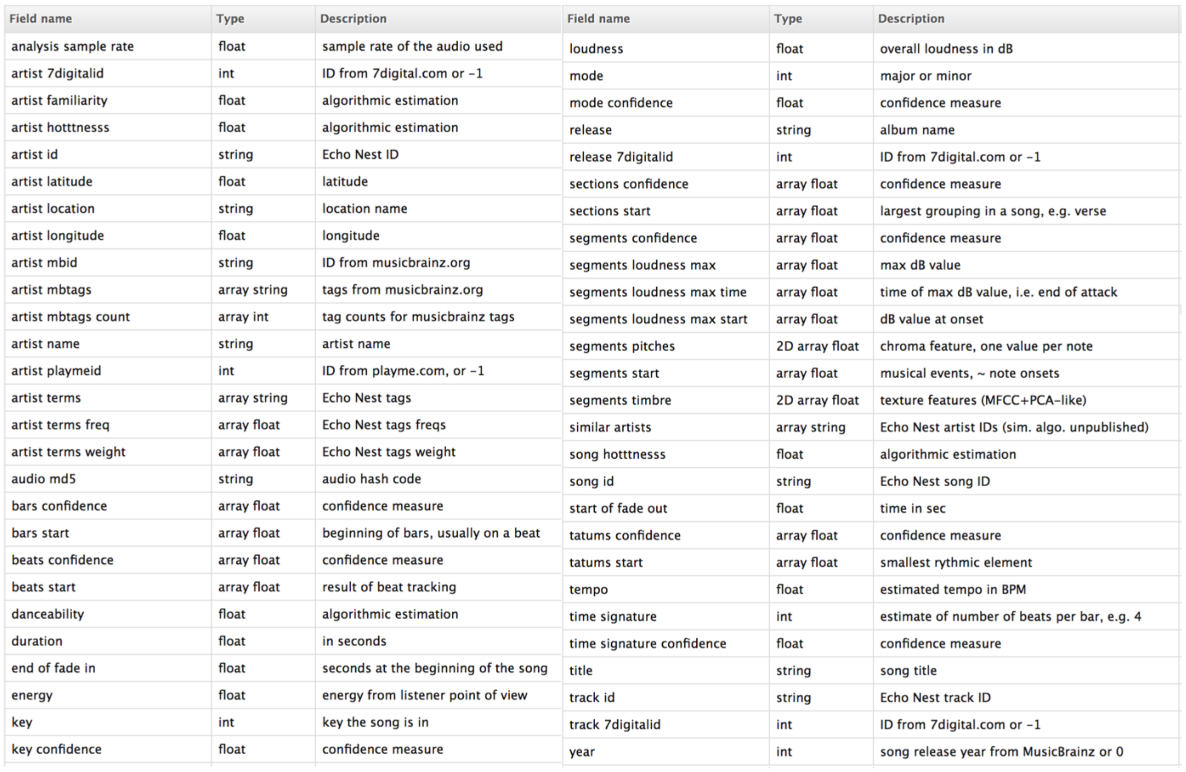
1.2. Dataset

This project involves two datasets. (1) 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 [2]. 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 us to correctly preprocess the data. Given one of the id's we 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. We 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. (2) Taste Profile Subset, a dataset consists of (userID, songID, playCount) curated by EchoN-



```
+--------------------+------------------+---------+
|                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 Dataset.



Field name	Type	Description	Field name	Type	Description
analysis sample rate	float	sample rate of the audio used	loudness	float	overall loudness in dB
artist 7digitalid	int	ID from 7digital.com or -1	mode	int	major or minor
artist familiarity	float	algorithmic estimation	mode confidence	float	confidence measure
artist hotttnesss	float	algorithmic estimation	release	string	album name
artist id	string	Echo Nest ID	release 7digitalid	int	ID from 7digital.com or -1
artist latitude	float	latitude	sections confidence	array float	confidence measure
artist location	string	location name	sections start	array float	largest grouping in a song, e.g. verse
artist longitude	float	longitude	segments confidence	array float	confidence measure
artist mbid	string	ID from musicbrainz.org	segments loudness max	array float	max dB value
artist mbtags	array string	tags from musicbrainz.org	segments loudness max time	array float	time of max dB value, i.e. end of attack
artist mbtags count	array int	tag counts for musicbrainz tags	segments loudness max start	array float	dB value at onset
artist name	string	artist name	segments pitches	2D array float	chroma feature, one value per note
artist playmeid	int	ID from playme.com, or -1	segments start	array float	musical events, ~ note onsets
artist terms	array string	Echo Nest tags	segments timbre	2D array float	texture features (MFCC+PCA-like)
artist terms freq	array float	Echo Nest tags freqs	similar artists	array string	Echo Nest artist IDs (sim. algo. unpublished)
artist terms weight	array float	Echo Nest tags weight	song hotttnesss	float	algorithmic estimation
audio md5	string	audio hash code	song id	string	Echo Nest song ID
bars confidence	array float	confidence measure	start of fade out	float	time in sec
bars start	array float	beginning of bars, usually on a beat	tatums confidence	array float	confidence measure
beats confidence	array float	confidence measure	tatums start	array float	smallest rythmic element
beats start	array float	result of beat tracking	tempo	float	estimated tempo in BPM
danceability	float	algorithmic estimation	time signature	int	estimate of number of beats per bar, e.g. 4
duration	float	in seconds	time signature confidence	float	confidence measure
end of fade in	float	seconds at the beginning of the song	title	string	song title
energy	float	energy from listener point of view	track id	string	Echo Nest track ID
key	int	key the song is in	track 7digitalid	int	ID from 7digital.com or -1
key confidence	float	confidence measure	year	int	song release year from MusicBrainz or 0

Figure 2: Million Song Dataset Schema

est consisting of 1,019,318 unique users, 384,546 unique MSD songs, and 48,878,586 triplets, see schema in Figure 1 [4].

2. Methods

Two types of methods that are commonly used in recommendation system are collaborative filtering and content-based filtering. We use both methods in order to take advantage of the similarities between songs and between users. Here is a pipeline of our recommendation system.

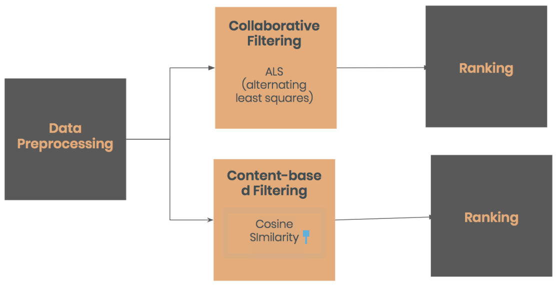


Figure 3: Pipeline of the Recommender

Preprocessed data are sent to the two models. Collaborative filtering uses the Taste Profile dataset only, while content-based biltering uses both two datasets. The recommendations from two models are combined and ranked to give the final recommendations.

2.1. 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. We say that they have similar "taste", and can make predictions (filtering) based on their tastes. 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, we have the problem of sparse data with respect to collaborative filtering because there are 1 million of songs. It is impossible for users to have listening records on all songs. If we 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, we choose matrix factorization to confront the problem of sparse data. In collaborative filtering, we use matrix factorization to decompose 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 we 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 matrix: 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 PySpark package (pyspark.ml.recommendation).

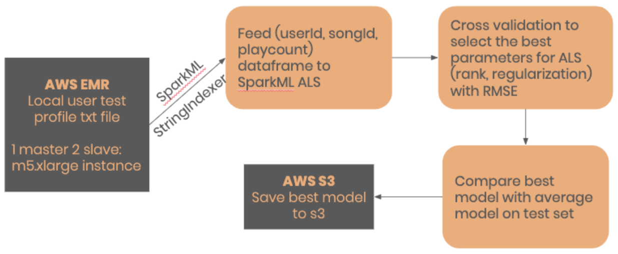

AWS EMR
Local user test profile txt file
1 master 2 slave: m5.xlarge instance
SparkML
StringIndexer
Feed (userId, songId, playcount) dataframe to SparkML ALS
Cross validation to select the best parameters for ALS (rank, regularization) with RMSE
Compare best model with average model on test set
AWS S3
Save best model to s3

Figure 4: Collaborative Filtering pipeline

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

2.2. 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. In the Million Song dataset, we have 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 previously.

One common way to measure the similarity between different song 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, we can recommend songs by ranking their cosine similarities between the user's previously listened songs. In the Million Song dataset, we 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. We proposed cosine similarity with adaptive features weights. That is, when computing cosine similarity, each feature dimension weights differently. Namely, for any two songs $song_1$ and $song_2$, 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, $\odot$ is the element-wise multiplication, x_1 and x_2 are the pre-processed feature vectors of $song_1$ and $song_2$ respectively. W is trained by maximizing the total cosine similarities of users' previously listened songs, which will be discussed further in Section 4.3.1.

To calculate this cosine similarity effectively and in a large-scale, we choose to use pyspark.ml.linalg package.

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 have high complexity. However, the Taste Profile dataset is relatively small. It was time-consuming in the step of hyperparameter tuning as it needs to build models and test several times.

On the other hands, cosine similarity can be highly parallelized as the cosine similarity need to be computed for any pairs of songs. In particular, to recommend songs for one user, we need to compute the cosine similarities between his previously listened song and all the candidate songs. Since cosine similarity can only be computed pair-wise, the computational complexity is $O(nm)$ where n is the number of user's previously listened 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, we use 1 master and 5 slaves of m5.xlarge instances on AWS EMR.

2.3. Metrics

In hyperparameter turning, we use RMSE (of plays) for ALS.

In evaluation, we use a separate test set where users have 30% of their listening history hidden. We 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 we 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 of 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).

Details can be found in the Result section.

3. Preprocessing and Computation

3.1. Data cleaning

The overall data cleaning pipeline is shown in Figure 5. Briefly speaking, the Million Song Dataset snapshot is attached to our 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.

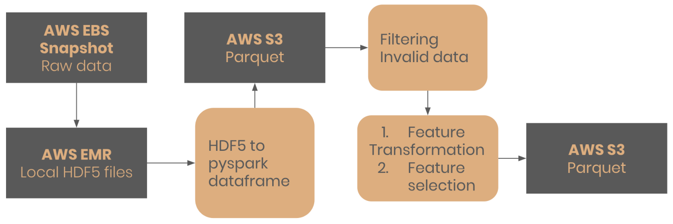


Figure 5: Preprocessing pipeline. We first load data from h5py and then converted it to a dataframe . Next, we filter out mismatched data. After doing some feature transformation and feature selection steps, we upload it onto AWS S3 with parquet format.

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 song 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 check the two found songs for consistency. If the two found songs are not the same, it means that mismatch occurs. Fortunately, Million Song Dataset website provides a list of mismatched songs and songs appearing in the list get filtered out from our data.

3.2. Feature transformation

Then let's take a close look at the dataset. 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 our project, we process different categories in a separate manner:

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

2. "string": To start with, we cleaned the original string data based on suggestions provided on the official dataset website. More specifically, we first did the entity resolution on both "song titles" and "artist names", and then we performed normalization on all string features. Next, we used 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 we 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 array, i.e. "array<float>" and "array<int>", the dimensionality relevant to the song length varies. What we do in our project is that we perform a subsampling strategy on this dimension, i.e. we divide each feature into 10 folds, and for each fold, we select only the most salient one by taking the maximum. In this way, we convert all $1 \times N$ features into 1×10. For 2-d array, such subsampling policy is not applicable; instead, we 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, we convert all 2-d array features with variable size $12 \times N$ into 12×10 and later, we unfold these 2-d array into 1-d vector with length 120.

4. "array<string>": The features of this data type are "artist mbtags" and. "artist terms". For simplicity concern, we throw away these 2 features since they are difficult to process with variable lengths and thus cannot be fit well with the built-in feature hasher.

3.3. Challenges

One typical challenge that is encountered is processing array<array<float>> data using PCA. The first issue is that PCA can be computationally expensive. Performing PCA over the features of one million songs can consume much time even computation is done in a distributed manner. The second issue 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 us 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.

3.4. Resources

We mainly used AWS EMR cluster because it provides an out-of-box set-up and configures all the system requirements such as Spark cluster, HDFS for us. In particular, we used 4 m5.2xlarge (1 master + 3 slaves) in the east-1 region for data cleaning. AWS S3 service is used as persistent storage for storing intermediate data because it can easily integrate with AWS EMR. As for the programing language, we mainly used python. We have used pyspark as our main framework for handling large-scale dataset. In order to handle h5df in which the data is originally stored, we used h5py and numpy. We also used sklearn and torch for implementing our machine learning algorithm. It cost approximately \$300 and 5.5 hours computational time in total.

3.5. Feature Selection

After performing all preprocessing steps on different data types, the next step we did is to 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 we should consider are scalar features of type float and int. What we do is that we first sample 10,000 samples randomly from the entire dataset (around 980,000 samples after removing mismatched data). Next, we compute the Pearson correlation over these 17 entries, which is formulated as:

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

We delete two features further ("year" and "energy") since they are all constant in our subsamples and thus informative. We plotted the heatmap over these features regarding their pairwise correlation coefficient in figure 6:

By setting the threshold as 0.7, we 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, what we do is that we only keep one feature

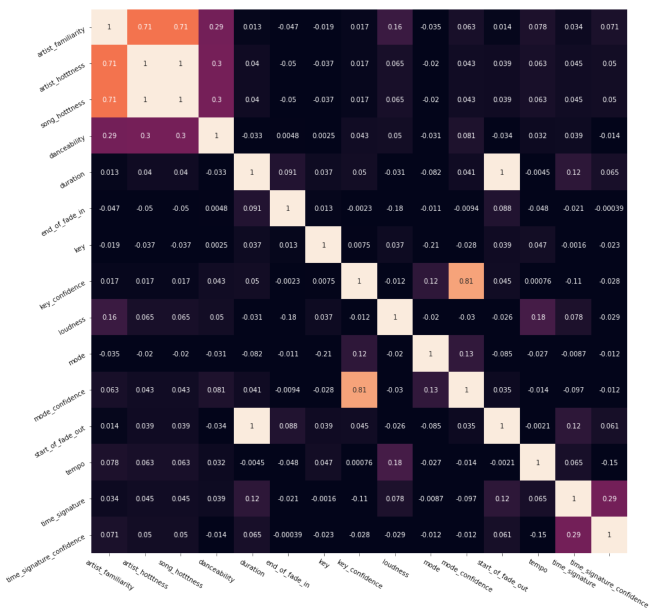


Figure 6: Heatmap over correlation coefficient of pairwise features

within each group and drop the rests. And finally, 11 / 17 scalar features are selected.

4. Results

4.1. Baseline

For this project, we devised a straightforward popularity-based baseline algorithm against which to compare our 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.

Rank	Song	Count
1	Fireworks	110479
2	Vespertine Live	90476
3	Now That's What I Call Music	90444
4	If There Was	84000
5	Only By The Night	80656
6	Waking Up	78353
7	Eine kleine Nachtmusik	69487
8	Karaoke	64229
9	Save Me	63809
10	Nova Bis	58610

Table 1: Top 10 Songs Listened to By Most Users

This method provide a solid basis for evaluating the effectiveness of our later work.

4.2. Collaborative Filtering

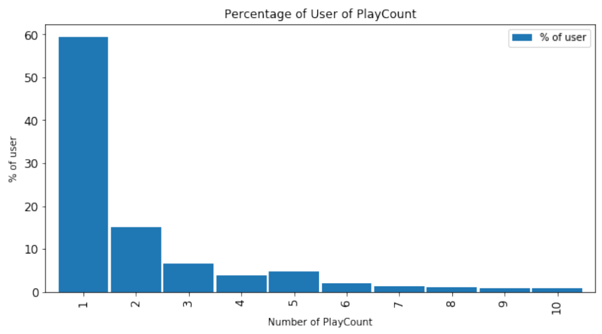


Figure 7: Distribution of User Play Count of User Taste Profile

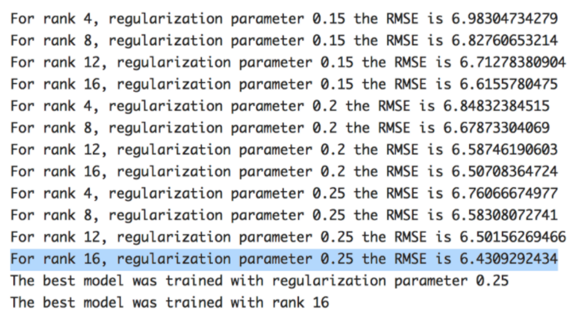
Collaborative filtering is a popular recommender system technique that can filter out items that a user might like on the basis of historical preference of a set of items by similar users. The core assumption for collaborative filtering is that the user who have agreed in the past tend to agree in the future as well. In our project, user preference is expressed by explicit rating: play count of a song by a user.

Recall from the introduction, we wish to answer the question: **which collaborative-based recommendation algorithm is suitable for solving production-sized problem?**

We apporoach this question by first exploring the user play count distribution from the user taste profile dataset. As seen in Figure 7, the distribution plot of user play count from the User Taste Profile dataset, 60% of the user from the dataset has only one listening history. While there are 1,019,318 unique users and 378,309 unique songs, the distribution of the data in user-item space is sparse. Since sparsity and scalability are two biggest challenges for collaborative filtering method, we used a matrix factorization approach to decompose the original sparse matrix to low-dimensional matrices with latent factors and less sparsity. The matrix factorization method used in this project is Alternating Least Square (ALS), which is one of the most popular scalable matrix factorization method for CF due to its robustness to popularity bias and item cold-start problem.

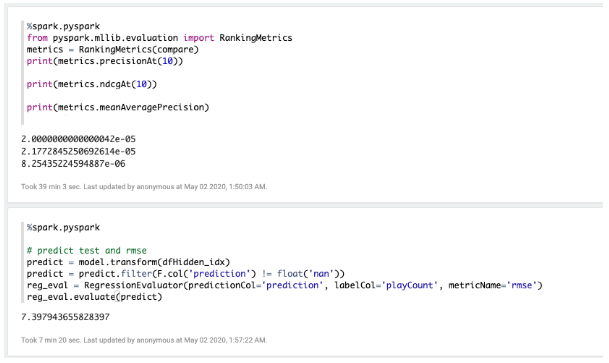
With the ALS method, we first performed grid-search to find the best hyperparameter configuration for ALS. As shown in the result in Figure 6, ALS has the best performance when rank = 16 and regularization parameter = 0.25.

Using this found hyperparamter configuration, we 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. We found that the regression metrics RMSE was exceptionally poor for the ALS method. The result is consistent on ranking metrics, as shown in Figure 9, 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.



```
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 8: Hyperparameter Search for ALS



```
%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

Took 39 min 3 sec. Last updated by anonymous at May 02 2020, 1:50:03 AM.
```

```
%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

Took 7 min 20 sec. Last updated by anonymous at May 02 2020, 1:57:22 AM.
```

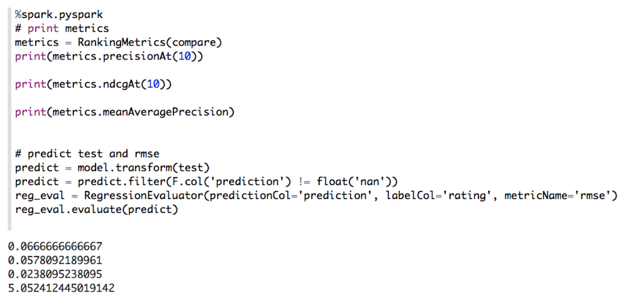
Figure 9: Evaluation for ALS, original

This lead us to consider the second question posed from the introduction: **what requirements need to be satisfied to achieve good performance on collaborative-based recommendation?**

Although the theory behind this method is compelling, there is not enough active user and song data for the algorithm to arrive at a good prediction. The sparsity may cause the ALS objective function to arrive at a sub-optimal local minimum, which results in poor RMSE rates.

In the attempt to arrive at a better result, we 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 preprocssing 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,



```
%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 10: Evaluation for ALS, removing inactive users and songs

which is an improvement from the previous ALS configuration (RMSE = 7.3979), and a improvement from the baseline (RMSE = 6.1882), indicating having informative user-item input is essential for collaborative-based recommendation performance.

4.3. Content-based Model

4.3.1 Training

In this section, we attempt to answer the third question posed from the introduction: **how to measure similarities between songs, and to design a suitable content-based recommendation method for this production-sized dataset?**

For content-based filtering, we use the preprocessed features mentioned in section 3, and then we design an algorithm to recommend new songs to user based on what they have listened before. First, the dataset is split into a training set and a test set. We use the training set to recommend songs to users and evaluate our recommendation on the test set. For simplicity, when one user has multiple songs available on the training set, we only select one song randomly and recommend multiple songs on the test set based on this song. Second, since this task is content based, we use both preprocessed audio and non-audio features. We assemble all scalar features into a large vector. For array features, we treat each entry as separate features, unfold them into 1-d array and concatenate with other features. In this way, the feature representation of each song is a vector of length 281. Third, we assume that different features should have varied importance, e.g. the artist similarity should not have the same weight as all 120 entries of an audio feature, so we also trained the coefficient for each feature dimension. Next, we make another assumption that, if one has listened a list of songs, all songs within this list should be similar based on their content; in other words, these songs should be close to each other in the feature space. We choose cosine similarity to model the distance between features, and the utility function we are trying to optimize is as follows:

$$U = \frac{1}{N'} \sum\nolimits_{N} \sum\nolimits_{x_1, x_2} \text{cosine similarity} \left(W \odot x_1, W \odot x_2\right)$$

where N' is the total number of all pairwise features, N is the number of users in the training set, W is the feature importance weight and x_1 and x_2 are features of two songs that this user has listened before. In other words, for each user, we want the averaged pairwise cosine similarities of his songs are as large as possible. In addition, we use a L1-regularizer to encourage sparsity: for instance, some aforementioned audio features are of size 12×10, where 12 is the MFCC dimension and 10 is the PCA-ed time dimension. It is likely that some entries are less important than other entries; with L1-regulaizer, we encourage the coefficient of some audio features to vanish. Also, by combining with feature selection steps in section 3, we include both automatic and manual feature selection steps in our project. We optimized this utility function with Pytorch and use the returned coefficients along each feature dimension to do the recommendation.

4.3.2 Evaluation

We evaluate our model with recall. Suppose that for one user, we select one song from his 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, we 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, we are only interested in model’s performance on recover the ground truth (a.k.a recall). The results are in figure 11. It can be shown that even with a very high threshold (0.9), there are around half of the ground truth data are recalled, which implies that our model can do a good job on recommendation.

4.4. Discussion

In this section, we attempt to answer the last question posed from the introduction: **How to measure recommender system performance?**

We implemented two song recommendation models. The first is a matrix factorization-based collaborative filtering model. We used the Alternating Least Square (ALS) algorithm implemented in Apache Spark ML. ALS is built for large-scale collaborative filtering problems, and is good at solving scalability and sparseness of the data. The second model is content-based, which utilizes song metadata and recommends songs based on song similarities.

Our results show a generally low performance in terms of precision produced by the song recommendation system.

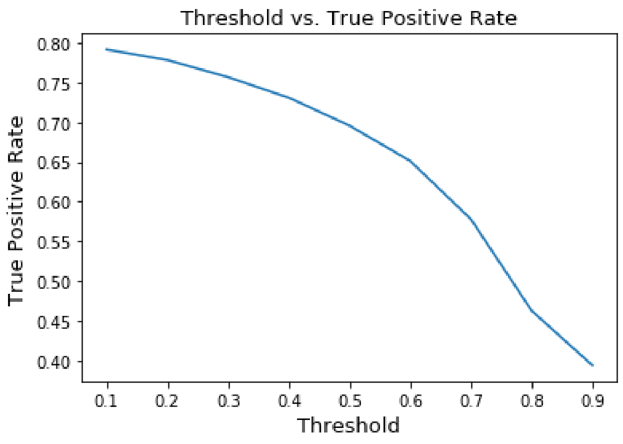


Figure 11: Content based recommendation recall vs. threshold

This is expected as most related works revealed a lack of correlation between users' listening history and their song preferences in the data set [3]. This also makes us question whether precision score is the best evaluation metrics for our 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 [1]. 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). Therefore, we argue that recall could be a better evaluation metrics for this particular problem, in that we are interested in finding out how effective is our model in recovering the ground truth. We demonstrate the effectiveness of our content-based model by plotting its recall score against increasing similarity score thresholds for recommendation.

We believe that the results can be improved in several ways. First, we can experiment with memory-based collaborative filtering methods, which identify either similar users or similar songs on the basis of play count. A lot of related works show that user-based collaborative filtering show very promising results [1, 3]. One challenge we might face with memory-based collaborative filtering is its lack of the ability to handle sparse datasets and scale to larger datasets. Second, we can also experiment with incorporating more features, choosing different combinations of features, or adjusting weight values for the chosen features for the content-based model. For example, we can experiment with only audio features or only non-audio features for modelling, and investigate which type of features contribute more to model performance. In addition, while most researchers focus on optimizing individual models, we think an alternative approach would be combining the results from different models to produce an optimal set of recommended songs. We hypothesize that content-based models will have better performance for users with smaller listening history, whereas collaborative filtering models will have better performance for users with larger listening history. By combining the two methods in a way that stresses each one's strengths, we expect the hybrid model will result in much better performance.

References

[1] Fabio Aiolli. A preliminary study on a recommender system for the million songs dataset challenge. volume 964, 01 2013.

[2] Thierry Bertin-Mahieux, Daniel PW Ellis, Brian Whitman, and Paul Lamere. The million song dataset. 2011.

[3] Yi Li, Rudhir Gupta, Yoshiyuki Nagasaki, and Tianhe Zhang. Million song dataset recommendation project report. 2012.

[4] B. McFee, T. Bertin-Mahieux, D. Ellis, and G. Lanckriet. The million song dataset challenge. 2012.