10605 Mini Project Final Report

Fangwei Gao **Ruoxin Huang** **Zhixian Li**
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
{fangweig, ruoxinh, zhixianl}@andrew.cmu.edu

1 Introduction

1.1 Problem Statement

In this project, the problem we want to explore is song recommendation on a large scale setting. Given a user's past listening history and musical features of the whole Million Song Dataset, we want to predict the songs that the particular user would most likely try. Without doubt, song recommendation systems that can capture users' ears hold great value to many music streaming services like Spotify and Pandora.

To solve this problem, there are two major kinds of algorithms, which are content-based filtering and collaborative filtering. Content-based filtering analyzes the similarity between song pairs according to song features, and recommend songs to a user if new songs are similar to what the user liked before. On the other hand, collaborative filtering models users' listening history as a sparse matrix and uses matrix factorization to solve the problem. The idea behind collaborative filtering is that users with similar behavior would make similar decisions about songs to listen to in the future.

1.2 Dataset

1.2.1 Million Song Dataset (MSD)

The Million Song Dataset is a large collection of extensive meta, audio and tagging information of one million music tracks, provided by The Echo Nest. The raw dataset is of 280GB large and is available on AWS snapshot.

There is also a MSD subset that only contains ten thousand songs that is directly downloadable. We leveraged this subset to test our code before running it on the full dataset.

1.2.2 Taste Profile Subset (TPS)

The Taste Profile Subset contains 48,373,586 (userID, songID, playCount) triplets of 1,019,318 unique users and 384,546 unique MSD songs. However, roughly 5.7k mismatches of songID and trackID were reported and removed from the dataset in our prepossessing.

2 Methodology

Please check figure 1 in appendix for the whole system pipeline.

2.1 Content-Based Filtering

As discussed in the introduction, the key point in content-based filtering is how to measure song similarity accurately. We extracted features from MSD and used user-song interaction data from TPS

as labels for supervision. For a single pair of songs, if the two songs were listened by a user, we consider the two songs are similar.

2.1.1 Data Pre-processing

We used our common sense music knowledge to select features that we thought would be helpful in determining similarity between songs. We tried out these feature on the MSD subset and made modifications to our selection according to the test results. In the end, we selected the features as shown in table 1 in appendix.

Of all the features, specifically, `segments_confidence`, `segments_loudness_max` are 1D float arrays of size `song_specific_length`, and `segments_pitches`, `segments_timbre` are 2D float arrays of shape `(song_specific_length, 12)`. We compressed these feature values by taking average of size-10 chunks on axis 0, thus obtaining 1D size-10 arrays and 2D shape `(10, 12)` arrays. We then flattened these features and joined with the rest floating point features to form a size 281 `features` array, associated with each track's `track_id`.

Our data points are concatenated song pairs of size 562, and they are label either similar (positive) or dissimilar (negative). If two songs were both listened to by 2 or more users, we would consider these 2 songs as a positive song pair. A challenge we faced with this approach is that our data only provides positive data points. To obtain negative data points, we randomly sampled global song pairs and checked it against our positive song pairs to make sure they are not positive song pairs.

2.1.2 Song Similarity Model

When computing song similarity, we only consider a single song pair at a time. Instead of simply computing a distance measurement like cosine similarity, which would result in poor performance since features have different scale and how to normalize them is unknown, we considered modeling this problem as a binary classification. To solve this problem, we used a deep neural network that is more expressive given the large dimensions of our features. As such, we built a 12 layer feed forward neural network with Relu layers and Batchnorm layers in between linear layers. We used Cross Entropy Loss as our lost function and used the Adam optimizer. Given a song pair as input, our model outputs the probability that the given song pair is similar. During inference, we would use our model to compute the similarity between the user's listening history and all other songs to the obtain the most similar (with highest probability) songs to recommend to the user.

2.1.3 Metric

To measure the performance of our model, we used the test classification accuracy along with the confidence level. To obtain the test accuracy, we held out part of our training data as the test set. This metric can tell us exactly how well our model can detect similarities in song pairs, and how confident our model is in the prediction. With a high accuracy and high confidence, we can guarantee to recommend similar songs to users, which is our ultimate goal.

2.2 Collaborative Filtering

Matrix factorization is a state-of-art solution for collaborative filtering on a large scale setting. It turns a sparse user-item iteration matrix into a product of two lower dimensional matrices. We use Alternating Least Square (ALS) algorithm to perform matrix factorization with user-song play count data as implicit feedback.

2.2.1 Data Pre-processing

We first loaded the (userID, songID, playCount) triplets on a local machine to associate TSP songID with MSD trackID and assign integer ids to all users and songs. Next, we removed songID and trackID mismatches, split the dataset into training and testing dataset and uploaded all datasets to AWS S3 to perform training and hyperparameter tuning. It should be mentioned that we also removed triplets that have `playCount=1`, since we observed significant model performance boost after removing these data points. We believe that these triplets do not reflect true preference of users.

2.2.2 Matrix Factorization and Alternating Least Square (ALS)

As mentioned previously, matrix factorization works by decomposing the user-item interaction matrix into a product of two lower dimensional matrices, a user embedding matrix and an item embedding matrix. The whole matrix factorization process should minimize the difference between the original matrix and the reconstructed matrix measured using root mean square error(RMSE). ALS solves this optimization objective by fixing one matrix at a time and alternatively minimizing the reconstruction error and L2 regularization using gradient decent.

2.2.3 Metric

We used RMSE as the objective function for model training and hyperparameter tuning. MAP(Mean Average Precision) and NDCG(Normalized Discounted Cumulative Gain) at 10 were also used in final performance testing.

3 Computation

3.1 Resources and Costs

We chose AWS as our cluster provider due to its convenient integration with our dataset and pySpark. We stored the full dataset in an s3 storage, and created an EMR cluster containing 1 master machine and 3 worker machines for computation. At first we used m5.xlarge machines for 24 hours, and later resorted to m5.2 xlarge for another 8 hours. Along with storage, we spent roughly $18, which was well below our budget.

3.2 Languages and Tools

In the data pre-processing and feature extraction steps, we worked solely with Python and Jupyter Notebook (using JupyterHub on EMR clusters). We used `h5py` and `tables` library to parse song files in `.h5` format; `pickle` to serialize intermediate data pre-processing results for check-pointing (which we will further explain in Challenges section); `boto3` to read large files stored on S3 such as original the MSD Dataset and also our data checkpoints; `torch` to build our neural network: and Python native `tempfile` libraries for data check-pointing and `itertools` library to perform negative sampling.

The most important framework we used throughout feature extraction and pre-processing is `Apache Spark`, specifically `PySpark` API, its `sql` and `ml` modules to apply to our `PySpark DataFrames` and `RDD`s. Here are some key pieces of logic that relied on `PySpark`:

1. to extract raw features from `.h5` files in parallel given the 1 million song files
2. to transform large 2D features such as `segments_pitches`, `segments_timbre`, into smaller chunked-average values and flatten with `pyspark.ml.feature.VectorAssembler`
3. to compute data-points by pairing two tracks and their features for our content-based model.

We used `ALS` from `pyspark.mllib.recommendation` to build the collaborative filtering model and `pytorch` to construct the neural network for content-based filtering. The collaborative filtering model was trained on the EMR cluster and the neural network was trained on Google Colab with GPU acceleration.

3.3 Challenges

Considering the expenses of EMR clusters, we first worked with the MSD subset (10k song entries) locally as proof of concept and to have working data pre-processing code with `PySpark` API. The first challenge came when we struggled to set up Spark on Google Colab. The official Google Colab tutorial did not work properly because of version issues. We searched extensively before coming across a Stanford CS246 notebook that allowed us to finally set up Spark running on Google Colab. From there we were able to write runnable feature pre-processing code that worked on the 10k dataset

successfully. This decision to write runnable code locally first proved a valuable time saver, as it eliminated any code bug issues when we started working with EMR clusters.

One of the biggest challenges we faced was kernel crashing in the mid of data processing: sometimes due to `OOM`, other times when we cancel a long-running cell and the session couldn't recover promptly. When this happened, we would need to restart the kernel, re-install all libraries (when short start-up time mattered), and re-run data-processing cells. Thus it was important to create checkpoints of key data pieces, which for us, included pairs of ids of tracks that had been listened to by the same user, `track_id` to flattened 1D `features` array mapping, data-points associated with positive labels, and self-generated data-points associated with negative labels. All these are in DataFrame format, so we could store them into S3 for check-pointing as parquet files. With intermediate data result saved, we avoided redundant re-computation on restarting the run time kernel, and also later when we had switched to larger clusters.

The most complicated piece of processing was probably to generate data-points with negative labels using negative sampling. With our local code copy, we could store the positive data-points in a bumpy array, generate random integers as row numbers to select tracks to compute negative-label data-points. However, we had ended up generating 10^8 size positive data-points in a data frame, with each row containing a string `track_id`, and a 562 floating-point number `features` array. Even an EMR cluster could not support storing all all that into an `numpy` array in memory, so we needed to come up with a more efficient sampling technique.

What we ended up with was to sample 200K points at a time for 20 iterations, store just the `track_id` (sans the `features` vector) in a native python list, and after each iteration save the python list in S3 with `pickle.dumps(str_obj)` and `boto3.client('s3').put_object(...)`. This way even if the kernel had crashed after `OOM`-ing at an iteration, we could recover the samples we've generated so far.

At this point we had only used a small amount of budget, so we decided to replace our current EMR cluster with a bigger one (m5.2XL machines instead of m5XL) to speed up the sample generation process. A bigger cluster easily made our per-iteration run time 5x faster(~5min to <1min), and also allowed us to have a 10^7 length Python list in memory when the smaller cluster had died when the list reached 10^6 lengths. From there the Python list we were able to generate an RDD, filter out bad samples and join with `track_id` -> `features` DataFrame to obtain the `features` vectors and create final data-points with negative labels.

4 Results

4.1 Content-Based Filtering

The metric we used to measure our model was test accuracy. Out of the 600k training data, we randomly held out 60K song pairs to be used as the test set. We trained our model for 10 epochs and reached above 99% training accuracy after 6 epochs, which was probably due to the large size of the training data. Our final test accuracy was 99.23%, with a average confidence of over 0.8. Therefore, we are very confident that our model can detect similarities between song pairs, and make excellent song recommendations.

4.2 Collaborative Filtering

We used 10% of the training dataset as a validation dataset for tuning two hyperparameters, `regParam` and `rank`, using grid search. `regParam=0.1` and `rank=22` gave us the best RMSE of 6.83. Then, we trained the model using all training data points and measured MAP and NDCG@10 on the testing dataset. The testing dataset have 50% of its user history visible to the model in training. We used the model to generate top 10 song recommendations for testing users and compared these recommendations with the holdout 50% user history in the testing dataset. The final NDCG@10 is 0.011 and MAP is 0.005. Please check figures in appendix for specific model configuration and evaliation results.

We think that the final performance is not very good in terms of NDCG and MAP. Further removing inactive users and unpopular songs should be able to improve our performance significantly as observed in our experiments.

References

[1] R. Bell, Y. Koren and C. Volinsky, "Matrix Factorization Techniques for Recommender Systems" in Computer, vol. 42, no. 08, pp. 30-37, 2009.

Appendix

Feature Name	**Type, Shape**	**Description**	**Post-processing Type, Shape**
artist_familiarity	float	algorithmic estimation	float
danceability	float	as titled, algorithmic estimation	float
duration	float	duration of the track in seconds	float
end_of_fade_in	float	as titled, at beginning of the track	float
key	float	key the track is in	float
key_confidence	float	as titled, confidence measure	float
loudness	float	overall loudness of track in dB	float
mode	int	major or minor	float
segments_confidence	array float, (var-len,)	confidence associated with each segment of the track	array float, (10,)
segments_loudness_max	array float, (var-len,)	max loudness during each segment	array float, (10,)
segments_loudness_max_time	array float, (var-len,)	time of the max loudness during each segment	array float, (10,)
segments_pitches	array float, (var-len, 12)	chroma features for each segment (normalized so max is 1)	array float, (10, 12)
segments_timbre	array float, (var-len, 12)	MFCC-like features for each segment	array float, (10, 12)
tempo	float	tempo in BPM	float
time_signature	int	estimate of number of beats per bar, eg. 4	float
time_signature_confidence	float	as titled, confidence measure	float

Table 1: Feature Table

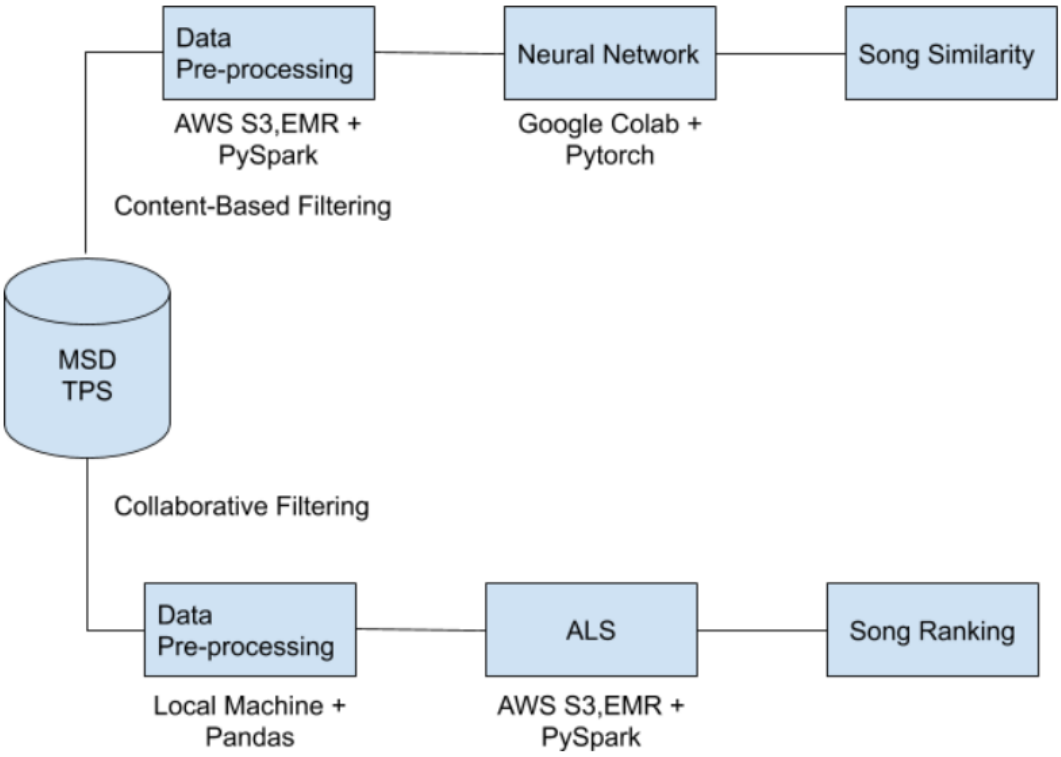

Figure 1: System Pipeline

```
import time
for regParam in range(10, 25, 3):
    for rank in range(10, 25, 3):
        als = ALS(maxIter=10,
          regParam=regParam/100,
          rank = rank,
          userCol="userId",
          itemCol="trackId",
          ratingCol="count",
          coldStartStrategy="drop",
          implicitPrefs=True)
        model = als.fit(train)
        val_preds = model.transform(val)
        evaluator = RegressionEvaluator(metricName="rmse", labelCol="count",
                                        predictionCol="prediction")
        rmse = evaluator.evaluate(val_preds)
        print("regParam: {}, rank: {}, rmse: {}".format(regParam, rank, rmse))
```

```
als = ALS(maxIter=10,
          regParam=0.1,
          rank = 22,
          userCol="userId",
          itemCol="trackId",
          ratingCol="count",
          coldStartStrategy="drop",
          implicitPrefs=True)
model = als.fit(all_train.union(test_history))
```

Figure 2: Model Configuration and Hyperparameter Tuning

```
In [38]: metrics.ndcgAt(10)

0.011292753922149153

In [40]: metrics.meanAveragePrecision

0.004833333333333334
```

Figure 3: Evaluation Results