Scalable Computing: Music Recommendation System

Yuying Chen (S3421902)
Zhuoyun Kan (S3346935)
Bogdan Petre (S3480941)

Group 6

April 1, 2018

1 Project Description

The main purpose of this project is to provide a recommendation list of songs based on the given song. It is first done by the batch process, extracting features of each song from a large historical playing dataset by users. Then, the streaming part takes the query song to predict a recommendation list of songs based on their similarities of extracted features.

2 Architecture

A high-level overview of the system architecture can be seen in Figure 1. The whole system mainly contains two parts based on Apache Spark, namely batch and online processing. First, the raw dataset from Last.fm[1] is preprocessed and stored in MongoDB. In the batch processing, a parallel algorithm using matrix factorization is developed to train using the whole dataset and the trained model is stored back in the database. Then, a predict function is called based on a given song name and the recommendation results are given as a text file and stored in the database as well.

Furthermore, an online streaming part is developed to deal with the incoming new data. The new data is coming from a Message Queue using Kafka. Once the consumer subscribes to a topic, it calls a predict function and uses the previously trained model to find similar songs. Then, the recommendation results are stored back in the database.

3 Components

The main components of the system are detailed in this section. In general, there are three main parts in this recommendation system, the dataset, the batch algorithm and the streaming process. The raw dataset is more than 2GB in size, containing 992 users. The batch algorithm is developed using factorization matrix and designed to be performed in parallel. The results are given based on the similarity between different items. Besides, the streaming process is developed using Kafka to deal with the query message queue.

[1] https://labrosa.ee.columbia.edu/millionsong/lastfm

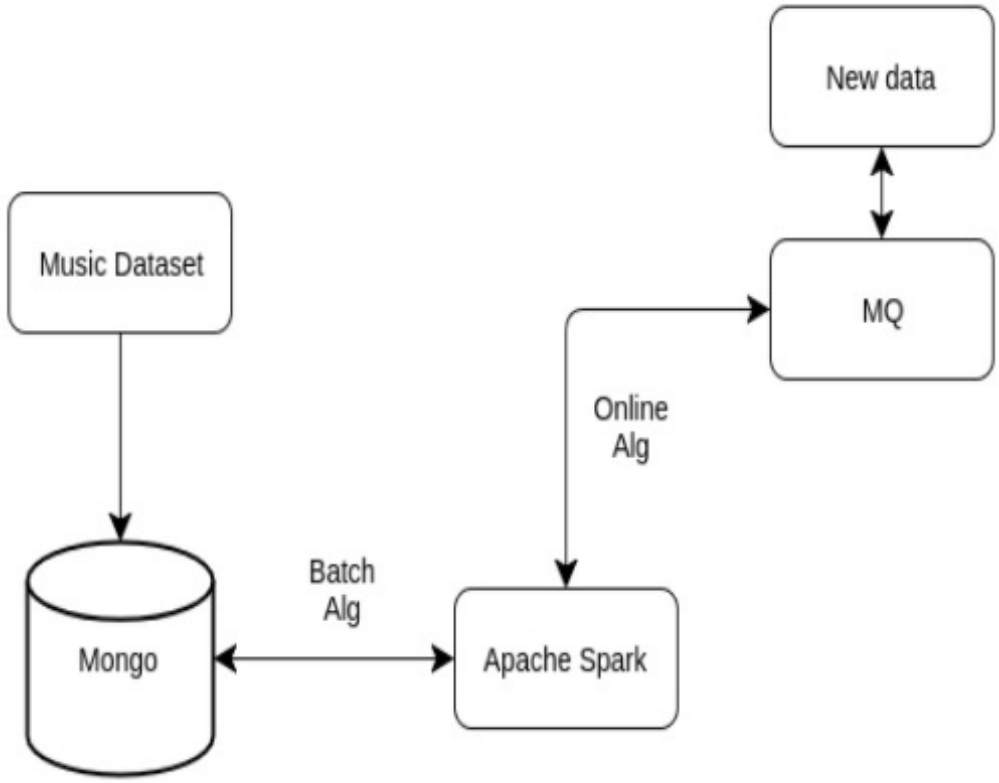

Figure 1: An overview of the system architecture

3.1 Dataset

The music dataset used in our project was collected from Last.fm API (till May, 5th 2009) for nearly 1,000 users. It stores 19,150,868 <user, timestamp, artist, song> tuples. However, the batch algorithm used in the project computes based on <user, song, playCount> tuple, so we preprocessed the initial dataset to a new version which contains 4,467,853 tuples. The size is also reduced from more than 2.0 GB to about 300 MB.

The link to initial archived dataset: click

After the preprocessing, the dataset was stored in MongoDB for later process of batch algorithm. The results of batch algorithm and the recommendation lists are also stored in MongoDB.

3.2 Algorithm

This algorithm refers to the batch algorithm used in our project. The ultimate goal is to find a list of candidates of the query song. In order to find the candidates, we chose to use the similarity of user behaviors across different songs. The main idea is that when two songs have more similar play counts among each user, they are more similar. Therefore, we have to create a large but sparse matrix of users and songs to compute the similarity when doing the recommendation. However, it is almost impossible to do it efficiently in this way, so the matrix factorization was used to reduce the complexity.

After matrix factorization, each song will have k factors (every song is stored as a vector) and each user will have k factors as well. When multiplying these factors, we can obtain the

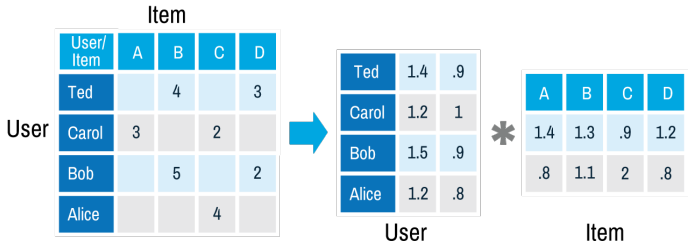

Figure 2: Concept about decomposition of original matrix (2 factors)

approximate play count of that song by the specific user. In other words, when two songs have similar vectors, their play counts by any users would be similar. The batch algorithm handles the matrix factorization and the streaming algorithm uses the song matrix to find top similar candidates for the query.

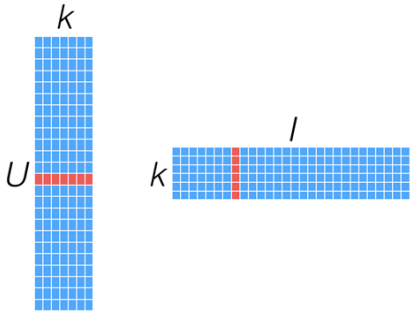

Figure 3: Find the approximate play count by User and Item vectors (7 factors)

The code of matrix factorization contains the map and reduce steps in the function "train" in the "Recommend.scala" file. For example, the decomposition step of creating tempSongMatrixRDD has a map step to compute a temporary dense matrix of a song from one user, and a reduce step to sum the matrix by the key of songs. The steps of map and reduce are processed with 10 iterations as recommended by Alternating Least Square (ALS) to minimize the error of approximation.

Since the first time we implemented the batch algorithm from Spark Mllib, we used RDD most of the time. We found it give us more room for operations, so we kept using RDD even we decided not to use the Spark Mllib but the current algorithm.

3.3 Streaming

We used Kafka as message queue to accept all query songs from different sources. A consumer will subscribe to the topic to get songs to predict. Currently, it only takes one song to find a recommendation list, but it can also be extended to take multiple songs together to give a recommendation list.

The streaming algorithm retrieves the vector representing the query song and computes its distances to all other songs. If the distances are smaller, it means the user behaviors of those songs are more similar to the query song. Instead of cosine similarity, we chose to use Euclidean distance to find the candidates, since we hope to keep the magnitude of play counts as well as song's popularity instead of pure user behaviors. After the prediction, the recommendation list will be saved to MongoDB.

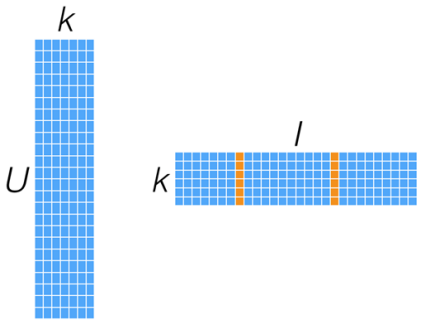

Figure 4: Find similar songs by vectors from the item matrix (7 factors)

4 Deployment

This section describes the way in which our system is deployed.

4.1 Spark Cluster

With the aid of Spark cluster, we are able to use the mechanism that Spark master uses to handle the tasks among all connected Spark workers. It lets the application process larger music datasets When there are more workers in the cluster, which increases the scalability.

4.2 Docker

We used Docker to containerize the application. Services are declared in a Docker Compose file and Docker containers are used for the different components: Mongo DB, Kafka, Spark Master, Spark Worker.

4.3 Docker-Compose

Docker-Compose is a tool for defining and running applications formed by multiple containers. Inside the docker-compose.yml file, all the services are declared, together with the Docker images, volumes, networks and links between services. In the docker-composed file, three services are declared for MongoDB, Spark master, and Spark worker respectively. Note that we only need to declare 1 service per component and then, when bringing up the project, we simply use `--scale spark-worker=2` to scale a certain service (in this case the Spark worker).

4.4 Makefile

We use command line arguments so that we don't hardcode IP addresses. In the Makefile, several commands are defined to run different components and deploy the whole application. For MongoDB, four commands are set to create, stop, start, and remove its docker image. For Spark, two commands are set for master and worker respectively. Also for Kafka, two commands are used to create Kafka image and Kafka topic.

Furthermore, for the whole application, it is composed, assembled and then deployed to the local cluster as a jar file simply using 3 commands. When deploying to the cluster, the IP addresses

of the Mongo container and Kafka container are passed as command line args, so that we do not hardcode IP addresses.

5 Results

The recommendation lists of all tested songs we input look quite correct, however, there are too many songs we are not familiar with. We can only judge the results from the artist names whether they are singers with similar styles or the songs are from the same artist.

One major bottleneck should be the streaming part. It takes about 30 seconds to finish the whole process to predict the recommendation list of a query song. It can only process next prediction when the first one is finished. It should be able to take multiple songs for their own recommendation lists at the same time. Besides, the step of comparing the similarity needs to retrieve matrix of all songs from the database, therefore it may retrieve the matrix every time of doing recommendation, which should be reused in our application.

Another bottleneck could be the step of loading the historical data from MongoDB, since the size of the dataset and the number of rows may be huge. It may need a long time to load all data but it should be able to be solved by building a mongo cluster to increase the loading speed.

```
Track Name: Coldplay|||God Put A Smile Upon Your Face
Recommendation List: Coldplay|||God Put A Smile Upon Your Face,
Radiohead|||Nude,
Oasis|||Stop Crying Your Heart Out,
The Ronettes|||Be My Baby,
Oasis|||Supersonic,
The Police|||Every Breath You Take,
The Killers|||Bones,
Coldplay|||Politik,
Travis|||As You Are,
Stevie Wonder|||Superstition

Track Name: Scorpions|||Send Me An Angel
Recommendation List: Scorpions|||Send Me An Angel,
The Osmonds|||Crazy Horses (Original Version),
Bryan Ferry|||Cruel,
Pet Shop Boys|||Hit And Miss,
Julien Baer|||La Folie Douce, High School Musical Cast|||You Are The Music In Me,
Body Rox Ft. Luciana -|||Yeah Yeah,
Le Vene Di Lucretia|||La Vestizione,
Sonic Youth|||Inhuman,
Kaki King|||The Footsteps Die Out Forever

Track Name: Guns N' Roses|||Attitude
Recommendation List: Guns N' Roses|||Attitude,
Placebo|||Haemoglobin,
Brakes|||Heard About Your Band,
Vnv Nation|||Interceptor,
The Duke Spirit|||Every Cell Can Tell,
Jan Delay|||Raveheart,
K-Os|||Sunday Morning,
Bob Marley|||Touch Me,
Håkan Hellström|||Den Fulaste Flickan I Världen,
Madvillain|||Meat Grinder
```