ECE 4813: Cloud Computing

Spring 2015

Team 7: Project Report

Directed Acyclic Graphs with Apache Spark

Members:

Monodeep Kar

Vinish Chamrani

Akshay Phadke

Daniel Morton

Dylan Slack

Zeheng Chen

1. Abstract

User-specific recommendations have become a frequent feature in commercial products in the last several years. Recent advances in data processing and analytics have led to increasingly accurate customized recommendations based on a user’s history of purchasing products or consuming content. On the content consumption side, prominent examples include Netflix and Pandora suggesting future multimedia content based on your browsing history. In particular, Pandora suggests the user to listen to specific songs based on the previous songs they have listened to.

While the application of our project was also to predict user song preferences based on listening history, our team took a different approach compared to the Pandora algorithms. We implemented a Directed Acyclic Graph in Apache Spark, using MapReduce, machine learning, and content-based filtering for song recommendations. Additionally, we used a subset of a very large (300 GB) data set of songs that is available online [1].

1. Problem Definition

Most of the song recommendation system use collaborative filtering, which requires access to user profile like the songs played by the user, the rating given by the user, etc. In absence of such information it becomes difficult to recommend songs to a user. So, to eliminate this problem we can use content based filtering which does not require information about the user, instead it leverages the information about the ietm profile which is songs in our case. So, we use the properties of songs to recommend new songs to the user. Also, we have made a public API so that people can improve the recommendation engine which is not the case with other recommendation engines. Also, to avoid latency problems we make use of Apache Spark which helps to process large dataset quickly.

1. Implementation
   1. Dataset Chosen

We chose a dataset of one million songs provided by columbia university. The dataset contains the following information about each song, artist metadata, song metadata and acoustic features. The dataset consists of a large no. of subdirectories each containing a HDF5 file. The wrapper code searches through the sub-directories to find any file with an extension (\*.h5) and applies functions from the hdf5\_getters.py file on the search result to extract various metadat. For eg., the function get\_artist(h5) is used to get the artist name of the song HDF5 file. After reading each fields, we store them into global arrays which are read later to store them into csv files.

Corresponding code: my\_total\_no.py

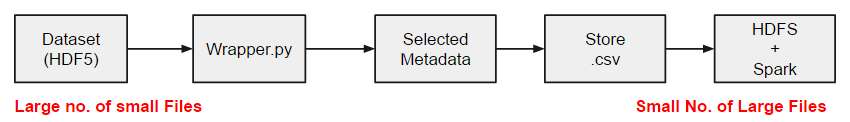


Fig. 1: Processing Dataset

* 1. Selecting Song Fields:

The most important aspect about song recommendation is selecting appropriate fields. The dataset consists of mainly three types of data. The artist and song metadata mainly contains different information about the artist and the song, like artist name, artist id, song name, song id etc. Different algorithmically computed values like danceability, loudness, energy represent acoustic attributes of a song which can be represented by a single number, either integer or float. The song is further broken down into time segments and different attributes of a time segment like timbre, loudness, mode, key are computed and stored into arrays of float or integer, which constitutes the third category of data. Here we have considered an array of four attributes (key confidence, mode confidence, loudness and tempo) as acoustic representation of any song. We will be use this vector of four elements for clustering purpose and recommend similar songs. Note that to improve the recommendation accuracy, we have to take into account the data from the third category and perform some signal processing on it to find similarity.

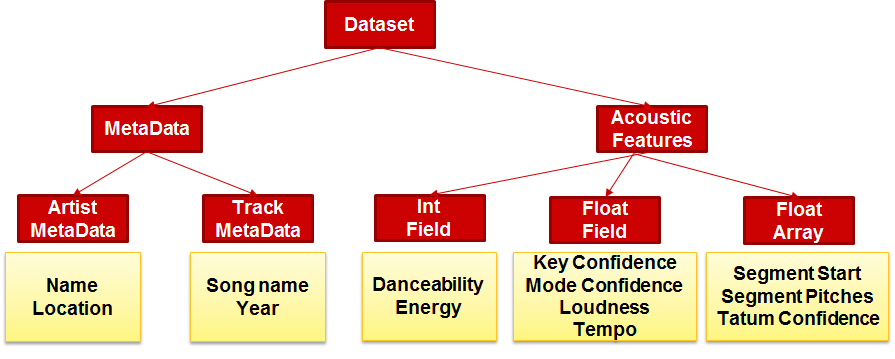


Fig. 2: Feature vectors used from Dataset

* 1. Machine Learning

To recommend similar songs we first wanted to classify the songs into various clusters. We divided our dataset into 10 clusters. Songs in one cluster were more correlated than songs in another cluster. This clusters served as a sort of genre which was missing from the dataset. To cluster the songs we used K-Means clustering. This clustering is available as a part of Spark’s MLlib[2].The implementation in MLlib has the following parameters:

* *k* is the number of desired clusters.
* *maxIterations* is the maximum number of iterations to run.
* *initializationMode* specifies either random initialization or initialization via k-means||.
* *runs* is the number of times to run the k-means algorithm (k-means is not guaranteed to find a globally optimal solution, and when run multiple times on a given dataset, the algorithm returns the best clustering result).
* *initializationSteps* determines the number of steps in the k-means|| algorithm.
* *epsilon* determines the distance threshold within which we consider k-means to have converged.

We configured KMeans.train with parameters as k = 10, maxIterations = 100, initializationMode = random, runs = 100.

Using above configuration, we used the feature vectors from the dataset to group the songs into various clusters. Now, to recommend for given song by the user we used Cosine Similarity and Euclidean Distance. The Cosine similarity tells how how much aligned two points in a cluster are while Euclidean Distance helps us to find out the distance of a point from the center. Using this, we calculated the cost function using the formula:

Cost function = [(Euclidean distance)2 - (1 - cosine\_similarity)2]

The songs in the cluster were sorted using this cost function for the song specified by the user. The top songs with minimum value of cost function were recommended as similar songs to the user. The results were stored in DynamoDB for each song along with their similar song. Following Fig. summarizes the above procedure.

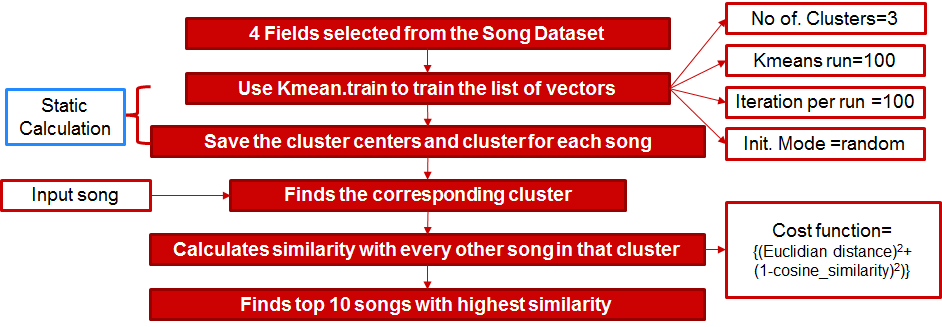


Fig.3: Architecture of recommendation engine

* 1. Architecture

After generating cluster information, we append the corresponding cluster information of each song into the csv file from which we read data for the machine learning part of the code. We also store the cluster centers into a pickle file. After we get the corresponding song-id as an user input from the command line, we look up the csv file containing cluster informations to find the corresponding cluster and load the pickle file to find the corresponding cluster center. The next step is to find every other song in the corresponding cluster apart from the user input song, and compute the euclidean distance between each song and the user input song. Note that, after clustering the search space is significantly reduced and this search can potentially be performed real time. The songs in the cluster are then sorted according to their euclidian distance with the user input song and top 10 songs are selected to display into the frontend API part.

The corresponding file is my\_code8.py

* 1. Generating “Top 10” Lists

sc = SparkContext("local", "Simple App")

client = MongoClient('mongodb://52.5.145.180:27017/')

print client.test\_database

db = client.main\_db

top\_s = db.charts

#artist\_name,artist\_id,song\_name,song\_id,artist\_hottness,song\_hottness,loudness,tempo,key,year

step1=sc.textFile("file:///home/ubuntu/output\_csv.csv")

step2=step1.map(lambda line: line.split(",")).filter(lambda line: len(line)>1)

step3=step2.map(lambda line: ((str(line[1].encode('utf-8')),str(line[0].encode('utf-8')))))

step4=step2.map(lambda line: ((line[1].encode('utf-8'),line[0].encode('utf-8'),float(line[4]),int(line[9])))).distinct()

for i in range (10)

inp\_year = 1990 + i

step5=step4.filter(lambda line: line[3]==inp\_year).map(lambda line: ((float(line[2]),(line[0],line[1],line[3]))))

step6=step5.sortByKey(False)

step7=step6.take(10)

step8=step2.map(lambda line: ((line[3].encode('utf-8'),line[2].encode('utf-8'),float(line[5]),int(line[9])))).distinct()

step9=step8.filter(lambda line: line[3]==inp\_year).map(lambda line: ((float(line[2]),(line[0],line[1],line[3])))).filter(lambda line: not math.isnan(line[0]))

step10=step9.sortByKey(False)

step11=step10.take(10)

post = {"year": sys.argv[1].encode('ascii'),

"tags": ["mongodb", "python", "pymongo"],

"date": datetime.datetime.utcnow()}

post["top-songs"]=[]

post["top-artist"]=[]

for i in step7:

artist = {"Name": (i[1][1]).encode('ascii')}

post["top-artist"].append(artist)

for i in step11:

song = {"Name": (i[1][1]).encode('ascii')}

post["top-songs"].append(song)

print post

post\_id=top\_s.insert\_one(post).inserted\_id

sc.stop

We wanted to create a recommendation of the most popular artists and songs belonging to the same year as the song being queried. In order to do this we first created a subset of the dataset by taking only the fields which we needed and stored them in a csv format. The attributes and the order in which they are stored is mentioned in the code as well.

artist\_name,artist\_id,song\_name,song\_id,artist\_hottness,song\_hottness,loudness,tempo,key,year

So for the top artist list compilation, we need the artist\_name, artist\_id andartist\_hotness. Using instance of SparkContext the csv file was read. The map transformation was used to split the comma separated values into words. Then we filter out the blank lines. The next step is to map the comma separated values into (artist\_id, artist\_name, artist\_hottness, year) and (song\_id, song\_name, song\_hottness, year). There are artists with multiple songs, so the distinct() transformation is used to find the unique artists so that they are not repeated while calculating top artists. We then filter out the artists and songs by the year, which we initially took as a system input, but replaced with a for loop to calculate the top artists and songs for each year. The next step is to change the keys from atist\_id and song\_id to artist\_hottness and song\_hottness which was done with another map transformation. However some of the songs had the song\_hottness value as NaN. So these values had to be filtered out. sortByKey() was then used to arrange the entries in descending order or hottness values and take() was used to return a list of top 10 songs and artists.These values generated were stored in MongoDB

|  |  |
| --- | --- |
| Python File |  |
| my\_total\_no.py, hdf5\_gettters | Wrapper code for reading hdf5 database, use appropriate functions from the hdf5\_getters file to generate the desired attributes of a song and store them into a csv file |
| my\_code4.py (Spark) | read the csv file generated from the wrapper and perform Kmeans clustering, and store back the cluster information back into a textfile |
| correction.py | parse the textfile stored by my\_code4.py and perform some text modification and store into a csv file |
| my\_code8.py (Spark) | read the csv file generated by correction.py and the user input song id from the command line, and finds the corresponding cluster and the top 10 similar songs in that cluster, stores the result into Mongo DB |
| toplist.py (Spark) | read the csv file generated from the wrapper and user input year from command line, finds the top song and top artist of that year and stores into Mongo DB |

Table 1: Source code files and their description

* 1. RESTful API

A RESTful (representational stateless transfer) web API was designed using Node.js, a popular javascript web framework. In particular, Express.js, a popular hypertext media library, was used with Node.js to route the API endpoints. The endpoints are ‘api/songs/ and ‘api/charts,’ which return all song and top yearly results in the database. The endpoints that return singular records are ‘api/songs/:song\_id’ and ‘api/charts/:year’, where ‘song\_id’ and ‘year’ can be replaced with either an integer or MongoDB ‘ObjectId.’ Currently, the only HTTP verb supported by the API is ‘GET’ requests. Because of the complexity of the project, the dataset is static and thus this is the only action available to the API. In the future, an entire suite of HTTP verbs could be implemented to fulfil REST’s CRUD (create, read, update, delete) operations.

* 1. Frontend GUI

For the front end, the team used the Handlebars.js semantic template tool, along with the Express 3 framework for Node.js. This allowed the team to use a minimal amount of readable code to create the desired views on the website.

The first view is the index page. This is simply a list of songs in the database, but all of the songs are hyperlinks that will provide more information when clicked. A screenshot of the index page is shown below.

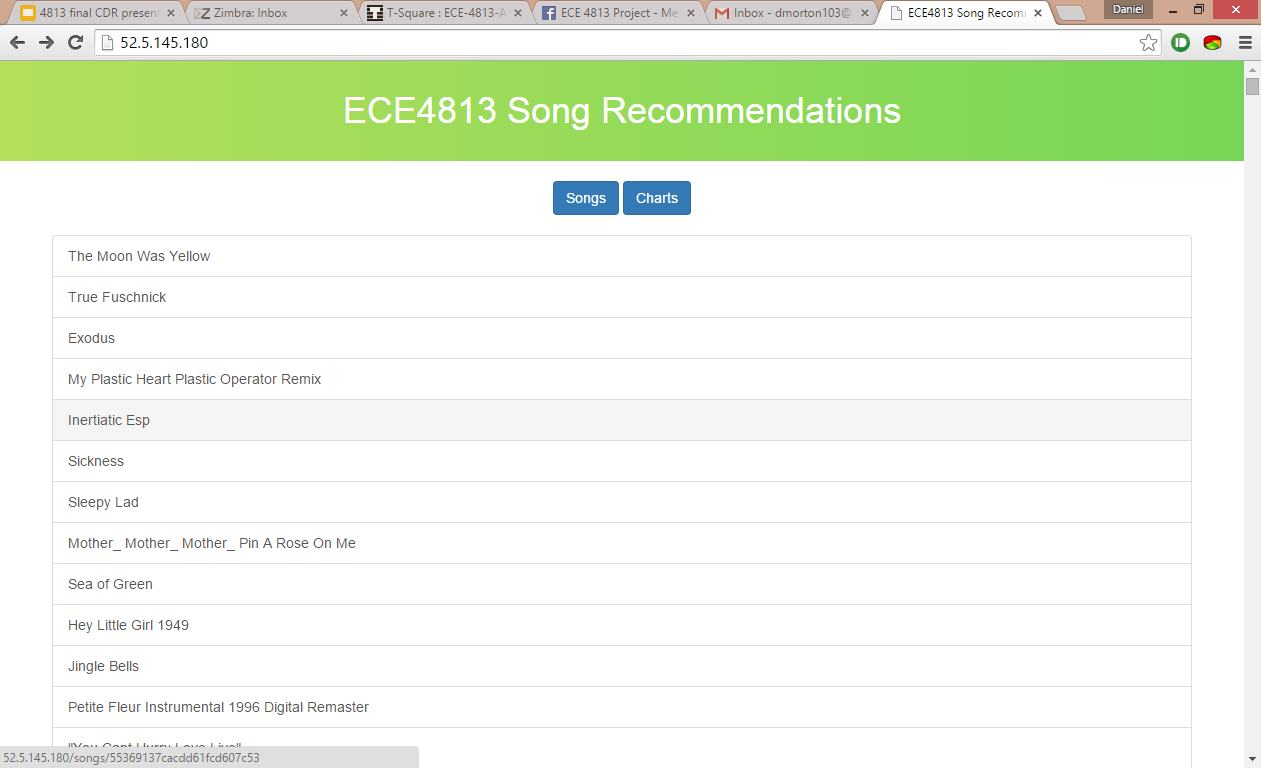


Figure 4. Index view

If an individual song is clicked, a link is created for that particular song, at web address <IP>/songs/<song\_id> . That brings up a webpage showing the details (metadata) for a specific song, including the title, artist name, and similar songs. An example of such a page is shown below.

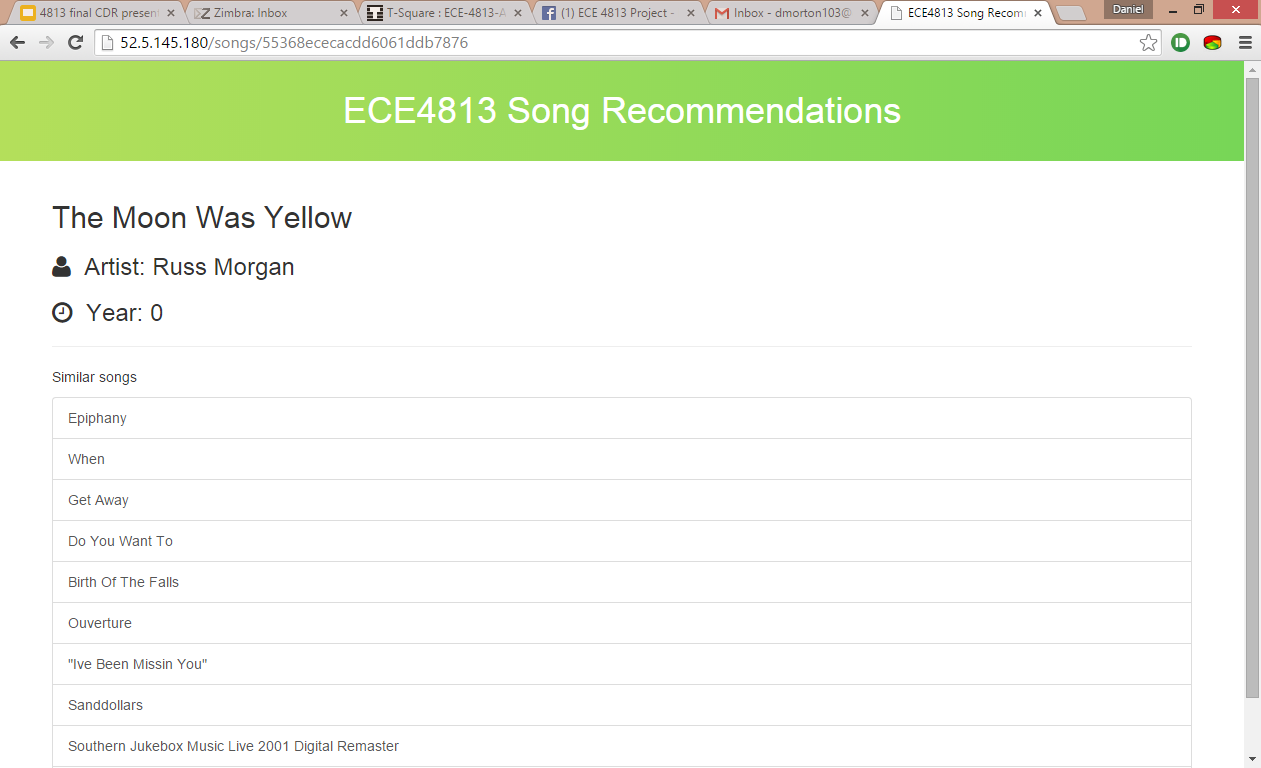


Figure 5. Individual song view

The last view is the “charts” view, which shows the most popular songs and artists for a particular year. This view is shown below (cut off for the screenshot after one year’s worth).

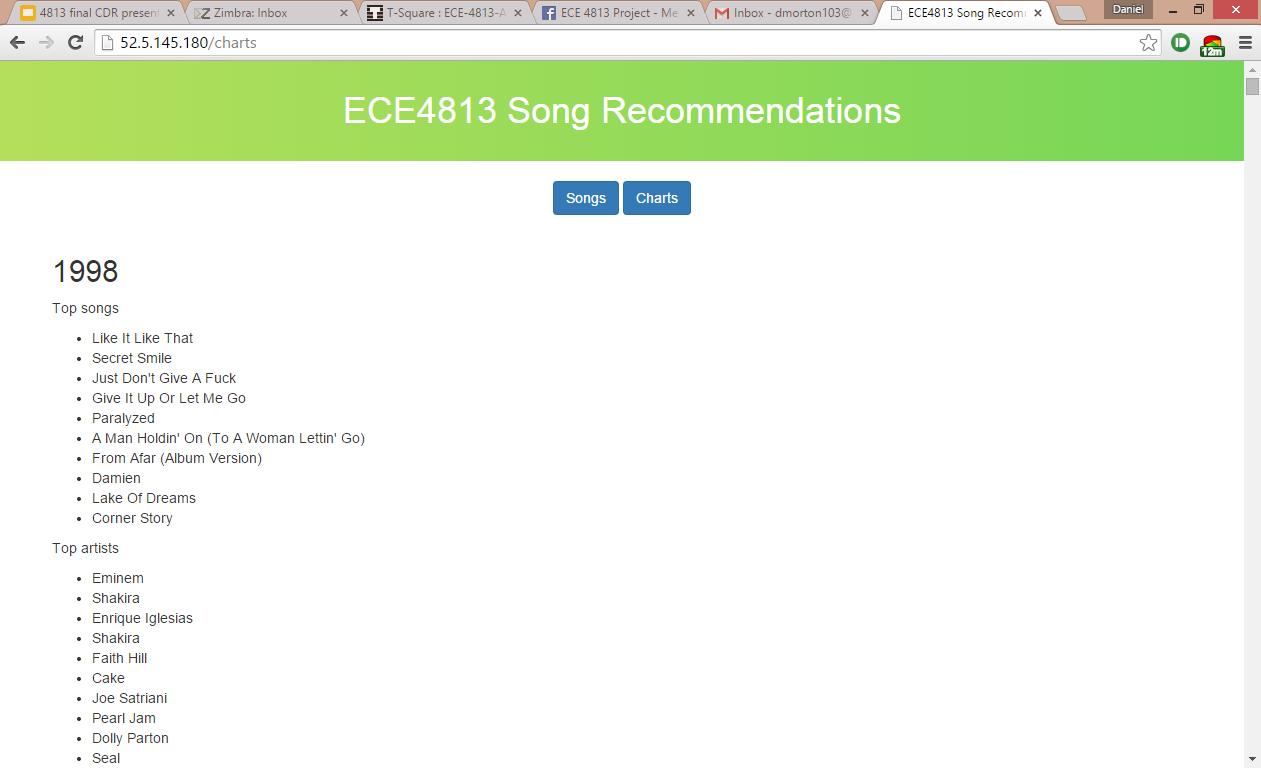


Figure 6. Charts view

* 1. Third-party Developer Demo
     1. API Used : GET /api/songs

AFHTTPRequestOperationManager \*manager = [AFHTTPRequestOperationManager manager];

[manager GET:@"http://52.5.145.180/api/songs/" parameters:nil success:^(AFHTTPRequestOperation \*operation, id responseObject) {

dispatch\_async(dispatch\_get\_main\_queue(), ^{

\_data = responseObject;

[self.tableView reloadData];

});

} failure:^(AFHTTPRequestOperation \*operation, NSError \*error) {

NSLog(@"Error: %@", error);

}];

* + 1. Work Flow

We developed an iPhone application to demonstrate how a 3rd party developer can use our song recommendation APIs to develop application.

When the app launched, it will grab all the songs data from the <http://52.5.145.180/api/songs/> . Then the app will populate the Table View with the songs data. When the user select a song in the table, it will show another Table View populated with the similar songs related to the song that the user select.

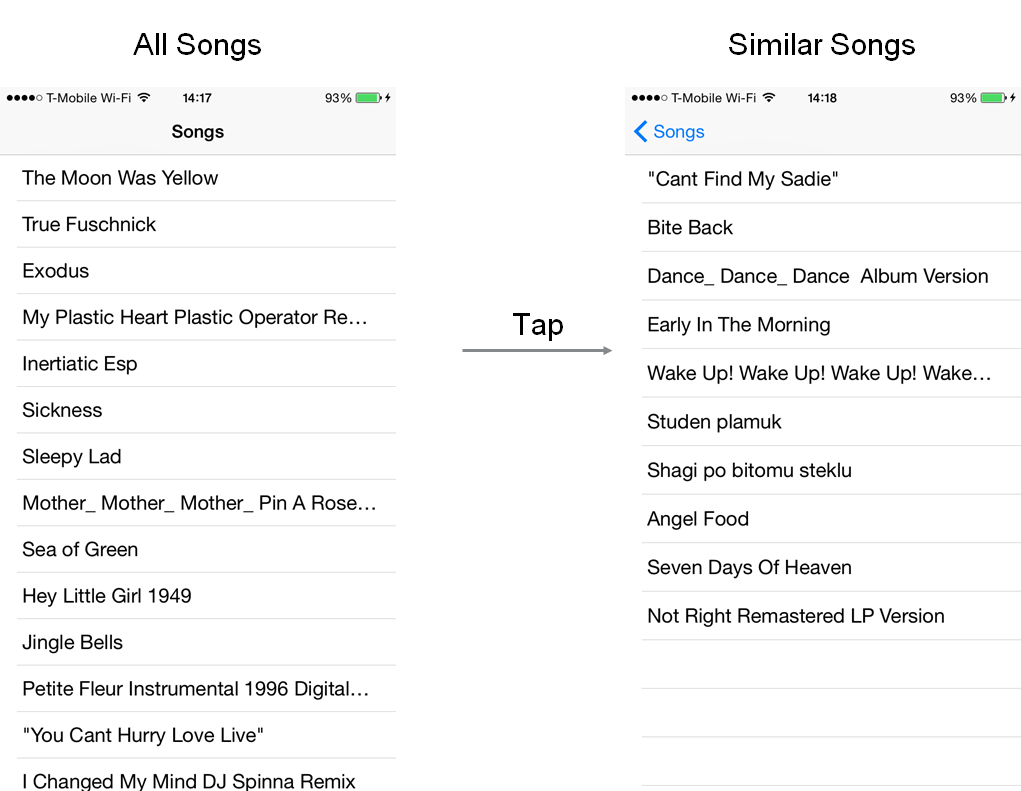


Figure 7. iPhone Application

* 1. Challenges

We faced several challenges while coding for the backend part. The original database came with a handful of data and picking a selective set among them was difficult. First we had to decide whether we will take integer, float or float array fields for clustering. We decided that clustering based on integer value might be inaccurate and for using the float arrays we need to perform some post processing for finding similarity. Hence for clustering purpose we chose 4 attributes from float fields. On of the challenges faced in the backend part was saving the intermediate results generated in the spark codes into an object file. After a ‘map’ transformation is used in Spark, a pipeline RDD file is generated which is difficult to store into a RDD. Hence we had no other choice that to store it in a textfile and hence for parsing that textfile for the later part of the code (my\_code8.py), we had to perform some string processing. Also we observed that Spark stops abruptly when its in-memory capacity gets full and it prevented us to run the program for all the 10000 songs. Another challenge was to learn Spark within a span of two weeks and finding out the exact transformations for processing the data. Also we would like to mention that the recommendation system is not full proof and the quality of recommendation can be significantly improved by taking into account other acoustic features, metadata and last but not the least user ratings.

1. Use cases and Results

We were able to process the dataset using Apache Spark and the result obtained from Apache Spark were successfully integrated with the front end to display the results. A user selects a song from the list of songs available to get the details about the song and also he can get suggestions about similar songs. Also, a user can view top songs and top 10 artists from the year 1990 to 2000. Also, using the API, developers can make iOS and Android apps to use the recommendation engine we developed to give information about songs and also recommend similar songs to the user. The screenshots in the above sections display all the results and use cases for our project.

In this way, we were able to do song recommendations in two ways:

1. By creating the genres, which were achieved by k-means clustering, we were able to recommend the songs which shared similar attributes to the song chosen by the user. This ensured that user gets similar songs.
2. Most often songs of the same year follow the same trend. And it is more likely that a popular song will be liked by the user as it resonates with everyone. We thus tried to recommend the top artists and top songs of the year which have a good chance of being liked by the user as well.

5. References

[1] Dataset, labrosa.ee.columbia.edu/millionsong/pages/getting-dataset

[2] MLlib, https://spark.apache.org/docs/1.1.0/mllib-clustering.html

[3] Apache Spark, https://spark.apache.org/docs/

[4] http://labrosa.ee.columbia.edu/millionsong/pages/code#python

[5] https://spark.apache.org/docs/latest/programming-guide.html#transformations