Building Music Recommendation Application

This application will that help predict what a user may like among a list of given items. This can bee viewed as an alternative to content search, as the application will help users discover new content . For example, Netflix will recommend movies based on historical view choices. Recommendation engines engage users to new services/movies and , can be seen as a revenue optimization process, in helping to maintain interest.

In this report I will demonstrate how to build a simple recommendation application, I will focus on music recommendations, and we use a simple algorithm to predict which songs users might like.

Goals

In this analysis, I expect to:

- Revisit or learn algorithms
- Build a simple model for a real user case: music recommendation system
- Understand how to validate the results

Steps

- Analyze data using Spark SQL and basic knowledge about the information.
- Define what is a sensible algorithm to achieve our goal: given the "history" of user taste for music, recommend new music to discover. I want to build a statistical model of user preferences to "predict" which additional music the user may like.
- Learn different ways to implement the algorithm, with the goal to illustrate the difficulties to overcome when implementing a (parallel) algorithm.
- Finally, I will focus on an existing implementation, available in the Apache Spark MLLib, which we will use to build a reliable statistical model.

One important topic I will cover in this analysis is how to validate the results obtained, and how to choose good parameters to train models especially when using a "muddy" library.

1. Data

Understanding data is one of the most important parts when designing machine learning algorithm. In this presentation, I will use the data set published by Audioscrobbler - a music recommendation system for last.fm. Audioscrobbler was founded in 2002 and is one of the first internet streaming radio sites. Audioscrobbler provides an open API for recording the number of times listeners play artists' songs.

1.1. Data schema

The datasets from Audioscrobbler only has information about events. It keeps track of how many times a user played songs for a given artist and the names of artists. This carries less information than a song rating.

Reading material:

- Implicit Feedback for Inferring User Preference: A Bibliography
- Comparing explicit and implicit feedback techniques for web retrieval: TREC-10 interactive track report
- Probabilistic Models for Data Combination in Recommender Systems

The data used in this presentation is available in 3 files:

- user_artist_data.txt: Contains app. 140,000+ unique users, and 1.6 million unique artists. It has 3 columns separated by spaces:

UserID	ArtistID	PlayCount
...	...	...

- artist_data.txt: provides the names of each artist by their IDs.

ArtistID	Name
...	...

- artist_alias.txt: The data in this file has 2 columns separated by tab characters. **Note:** The application submits the name of the artist being played and the name could be misspelled; i.e. "The Smiths", "Smiths, The", and "the smiths". This appears as three distinct artist IDs in the data set, even though they are the same. The artist_alias.txt will map artist IDs known misspellings or variations to the recognized Artist ID.

MisspelledArtistID	StandardArtistID
...	...

1.2. Data Exploration: Simple Descriptive Statistics

To be able to choose/design a suitable algorithm to achieve the analysis goal, you nee to understand data characteristics. To begin, import the necessary libraries to work with regular expressions, Data Frames, and other nice features of our programming environment.

For this analysis we are given the schema for the data we use, which is as follows:

userID: long int

artistID: long int

playCount: int

In [1]:
```
# Downloading dataset
!set -e

!mkdir -p dataset
!cd dataset
!wget http://www.iro.umontreal.ca/~lisa/datasets/profiledata_06-May-2005.tar.gz
!tar xvf profiledata_06-May-2005.tar.gz
!mv profiledata_06-May-2005/* dataset/
!rm -r profiledata_06-May-2005
```

```
--2018-11-23 06:24:25--  http://www.iro.umontreal.ca/~lisa/datasets/profiledata_0
Resolving www.iro.umontreal.ca (www.iro.umontreal.ca)... 132.204.26.36
Connecting to www.iro.umontreal.ca (www.iro.umontreal.ca)|132.204.26.36|:80... co
HTTP request sent, awaiting response... 200 OK
Length: 135880312 (130M) [application/x-gzip]
Saving to: 'profiledata_06-May-2005.tar.gz.2'

100%[======================================>] 135,880,312 25.1MB/s   in 4.8s

2018-11-23 06:24:31 (27.2 MB/s) - 'profiledata_06-May-2005.tar.gz.2' saved [13588

profiledata_06-May-2005/
profiledata_06-May-2005/artist_data.txt
profiledata_06-May-2005/README.txt
profiledata_06-May-2005/user_artist_data.txt
profiledata_06-May-2005/artist_alias.txt
```

In [2]:
```
!cat dataset/user_artist_data.txt | head
```

```
1000002 1 55
1000002 1000006 33
1000002 1000007 8
1000002 1000009 144
1000002 1000010 314
1000002 1000013 8
1000002 1000014 42
1000002 1000017 69
1000002 1000024 329
1000002 1000025 1
cat: write error: Broken pipe
```

In [3]:
```
import os
import sys
import re
import random
from pyspark import SparkContext
from pyspark.sql import SQLContext
from pyspark.sql.types import *
from pyspark.sql import Row
from pyspark.sql.functions import *

%matplotlib inline
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
from time import time

sqlContext = SQLContext(sc)
# Base directory path
base = "dataset/"
```

Using SPARK SQL, load data from `/dataset/user_artist_data.txt` and show the first 20 entries (

For this Notebook, from a programming point of view, we are given the schema for the data we use, wh

```
userID: long int
artistID: long int
playCount: int
```

Each line of the dataset contains the above three fields, separated by a "white space".

In [4]:
```
userArtistDataSchema = StructType([ \
    StructField("userID", LongType(), True), \
    StructField("artistID", LongType(), True), \
    StructField("playCount", IntegerType(), True)])

userArtistDF = sqlContext.read \
    .format('com.databricks.spark.csv') \
    .options(header='false', delimiter=' ') \
    .load(base + "user_artist_data.txt", schema = userArtistDataSchema) \
    .cache()

# we can cache an Dataframe to avoid computing it from the beginning everytime it
userArtistDF.cache()

userArtistDF.show()
```

```
+-------+--------+---------+
| userID|artistID|playCount|
+-------+--------+---------+
|1000002|       1|       55|
|1000002| 1000006|       33|
|1000002| 1000007|        8|
|1000002| 1000009|      144|
|1000002| 1000010|      314|
|1000002| 1000013|        8|
|1000002| 1000014|       42|
|1000002| 1000017|       69|
|1000002| 1000024|      329|
|1000002| 1000025|        1|
|1000002| 1000028|       17|
|1000002| 1000031|       47|
|1000002| 1000033|       15|
|1000002| 1000042|        1|
|1000002| 1000045|        1|
|1000002| 1000054|        2|
|1000002| 1000055|       25|
|1000002| 1000056|        4|
|1000002| 1000059|        2|
|1000002| 1000062|       71|
+-------+--------+---------+
only showing top 20 rows
```

How many distinct users do we have in our data?

In [5]:
```
allusers = userArtistDF.count()
print("All rows in database: ", allusers )
uniqueUsers = userArtistDF.select('userID').distinct().count()
print("Total n. of distinct users: ", uniqueUsers)
```

```
All rows in database:  24296858
Total n. of distinct users:  148111
```

How many distinct artists do we have in our data ?

In [6]:
```
uniqueArtists = userArtistDF.select('artistID').distinct().count()
print("Total n. of artists: ", uniqueArtists)
```

```
Total n. of artists:  1631028
```

What are the maximum and minimum values of column userID ?

In [7]:
```
MAX_VALUE = 2147483647
#showing the IDs which are invalid
userArtistDF[(userArtistDF.userID.cast("int") < 0)].show()
userArtistDF[(userArtistDF.userID.cast("int") > MAX_VALUE)].show()
#As the result, we don't see any invalid userID
#Otherwise, we can use function describe() of Spark MLlib to show the statistics
userArtistDF.select("userID").describe().show()
```

```
+------+--------+---------+
|userID|artistID|playCount|
+------+--------+---------+
+------+--------+---------+

+------+--------+---------+
|userID|artistID|playCount|
+------+--------+---------+
+------+--------+---------+

+-------+------------------+
|summary|            userID|
+-------+------------------+
|  count|          24296858|
|   mean|1947573.2653533637|
| stddev| 496000.5551820078|
|    min|                90|
|    max|           2443548|
+-------+------------------+
```

What is the maximum and minimum values of column artistID ?

In [8]:
```
#Finding min and max of artistID using function groupby() and max|min()
max_artistID = userArtistDF.groupby().max('artistID').collect()[0].asDict()['max(
print('Maximum value of artistID: ',max_artistID)
min_artistID = userArtistDF.groupby().min('artistID').collect()[0].asDict()['min(
print('Minimum value of artistID: ',min_artistID)

#Again, we can use describe() for short
userArtistDF.select("artistID").describe().show()
```

```
Maximum value of artistID:  10794401
Minimum value of artistID:  1
+-------+------------------+
|summary|          artistID|
+-------+------------------+
|  count|          24296858|
|   mean|1718704.0937568964|
| stddev|2539389.0924284603|
|    min|                 1|
|    max|          10794401|
+-------+------------------+
```

Maximum value of artistID:

Minimum value of artistID:

Based on the analysis we just discovered that there are a total of **148,111 users** in the dataset. Similarly, there is a total of **1,631,028 artists** in the dataset**. The maximum values of userID and artistID are still smaller than the biggest number of integer type**. No additional transformation will be necessary to use these IDs.

SPARK SQL provides a very concise and powerful method for data analytics compared to the usage of RDD. You can see more examples here.

We might also want to understand better user activity and artist popularity. Below is a list of simple descriptive queries that may helps us reaching these purposes:

- How many times does each user play a song? This might be a good indicator of who are the most active users of the service. Note: very active user with many listens may not necessarily mean the user is also "curious". For example a user could have played the same song several times.
-
- How many play counts for each artist? This is maybe a good indicator of the artist popularity. There are no dates associated with when the songs were played. We will need to be careful when determining popular artists.

How many times each user has played a song? Display 5 samples of the result.

```
In [9]: # Compute user activity
        # We are interested in how many playcounts each user has scored.
        userActivity = userArtistDF.groupBy('userID').sum('playCount').collect()
        print(userActivity[0:5])
        len(userActivity)

        [Row(userID=1000061, sum(playCount)=244), Row(userID=1000070, sum(playCount)=2020
        sum(playCount)=201), Row(userID=1000832, sum(playCount)=1064), Row(userID=1000905
Out[9]: 148111
```

We have total 371638969 play counts.

In average, every user plays 2509 times.

25% of the users have the play counts less than or equal to (<=) 204 times.

50% of the users have the play counts less than or equal to (<=) 892times.

75% of the users have the play counts less than or equal to (<=) 2800 times.

95% of the users have the play counts less than or equal to (<=) 10120 times.

99% of the users have the play counts less than or equal to (<=) 21569 times.

The result is plausible with the figure above.

About **7746 users (5.23%)** have the play counts less than or equal to (<=) **10 times**. These users have very little interaction with the system, so there is more difficult for recommending for these users than other users creating more impact in the system (have a certain number of playCount).

How many play counts for each artist?

Add in chart

Sum = 371638969

Mean = 227.855664648

Min = 1

Max = 2502130

Top 5 play counts: [1425942 1542806 1930592 2259185 2502130]

Sum top 5 artist play counts: 9660655

Percentage of top 5 artist play counts: 0.0259947309239

Play Count <=10 = 0.7486793605014751

Play Count<=1000 = 0.987435531456235

Answer :

Total 371638969 play counts. In average, PlayCount per artist is 227 times

Only 74.87% of the artists is played less than or equal to (<=) 10 times.

And 98.74% of the artists is played less than or equal to (<=) 1000 times.

We also have top 5 artist play counts: [1425942 1542806 1930592 2259185 2502130]. This accounts for 2.6% on overall number of playCount (5 out of 1631028 artists). The play count of top 5 artist is much higher than the mean. We can imply that the application can recommend the most-played artists to every user with this top 5 artists, and still get high performance.

Add bar chart to show top 5 artists In terms of absolute play counts.

These results seem reasonable.

- Previously, 98.74% of the artists have play count less than or equal to (<=) 1000 times. Because the top-5-artists are not sufficient to extract information about the data because they are the outliers and they should be cut out of data.
- The top-5-artists take about 2.6% of the total play counts, but 98.74% of the artists have the play count less than or equal to (<=) 1000 times. It appears some artists are played much more than other artists.
- This does not address the artist "disambiguation" issue.
    - Are the artist ID used referring to unique artists?
    - How do we make sure "opaque" identifiers point to different bands?
    - To determine whether a single artist has two different ids we can try an additional dataset to answer this question. This time, the schema of the dataset consists in:
        - artist ID: long int
        - name: string

1.3 Data Integration

Look at top 20 artists whose name contains "Aerosmith". Look at artists with the ID equal to 1000010 and 2082323. Are they pointing to the same artist?

In my opinion, they are pointing to the same artist. To ensure we are correctly answering the question we need to use an additional dataset "artist_alias.txt". This dataset contains the ids of misspelled artists and standard artists. The schema of the dataset consists in:

mispelledID ID: long int

standard ID: long int

In [14]:
```
customSchemaArtist = StructType([ \
    StructField("artistID", LongType(), True), \
    StructField("name", StringType(), True)])

artistDF = sqlContext.read \
    .format('com.databricks.spark.csv') \
    .options(header='false', delimiter='\t', mode='DROPMALFORMED') \
    .load(base + "artist_data.txt", schema = customSchemaArtist) \
    .cache()
artistDF.show(5)
```

```
+--------+--------------------+
|artistID|                name|
+--------+--------------------+
| 1134999|        06Crazy Life|
| 6821360|        Pang Nakarin|
|10113088|Terfel, Bartoli- ...|
|10151459| The Flaming Sidebur|
| 6826647|   Bodenstandig 3000|
+--------+--------------------+
only showing top 5 rows
```

Verify the answer of "Are artists that have ID equal to 1000010 and 2082323 the same ?" by finding the standard ids corresponding to the mispelled ids 1000010 and 2082323 respectively.

After evaluating mispelledID, the guess is correct since the artistID = "2082323" has standardID="1000010", meaning it represents the same artist

The misspelled or nonstandard information about artist make our results in the previous queries a bit messy. To overcome this problem, we can replace all misspelled artist ids by the corresponding standard ids and re-calculate the basic descriptive statistics on the "cleansed" data. The first step is to construct a "dictionary" mapping non-standard ids to a standard ones. This "dictionary" will be used to replace the misspelled artists.

Although there some advantages, explicitly creating broadcast variables is only useful when tasks across multiple stages need the same data or when caching the data in deserialized form is important.

With the "clean" data we can use it to redo previous statistic queries.

How many unique artists? 1568126 The number of artists is reduced from 1631028 to 1568126 after cleaning and standardizating the misspelled with Standard ID.

Who are the top-10 artists? Add chart

After removing all misspelled id there is little increase in playCounts of top-10-artists.

Here is the comparison on Play Count of top 10 artist before and after removing misspelled id:

Add % decrease

artistID=979: 2502130 -> 2502596.

artistID=1000113: 2259185 -> 2259825.

artistID=4267: 1930592 -> 1931143.

artistID=1000024: 1542806 -> 1543430.

artistID=4468: 1425942 -> 1426254.

artistID=82: 1399418 -> 1399665.

artistID=831: 1361392 -> 1361977.

artistID=1001779: 1328869 -> 1328969.

artistID=1000130: 1234387 -> 1234773.

artistID=976: 1203226 -> 1203348.

The chart shows Play Count dramatically decreases from the 1st to the 4th artist in top-10-artists, however from the 4th artist to 10th artist it only decreases slightly.

Who are the top-10 users? add the bar chart

How many different artists they listened to? add the bar chart

2. Build a statistical model to make recommendations

2.1 Introduction to recommendation application

In a recommendation application there are two classes of entities, which we shall refer to as "*users*"and *"items*". *"Users*" have preferences for certain items which must be inferred from the data. The data itself is represented as a preference matrix giving each user-item pair, a value representing what is known about the degree of preference of that user for that item. The table below is an example for a preference matrix of 5 users and k items. The preference matrix is also known as utility matrix.

\	IT1	IT2	IT3	...	ITk
U1	1	...	5	...	3
U2	...	2	...	...	2
U3	5	...	3	...	...
U4	3	3	...	...	4
U5	...	1	...	...	...

The value of row i, column j expresses how much does user i like item j. The values are often the rating scores of users for items. An unknown value implies that we have no explicit information about the user's preference for the item. The goal of a recommendation system is to predict "the blanks" in the preference matrix. For example, assume that the rating score is from 1 (dislike) to 5 (love), would user U5 like IT3 ? We have two approaches:

- Designing our recommendation system to take into account properties of items such as brand, category, price... or even the similarity of their names. We can denote the similarity of items IT2 and IT3, and then conclude that because user U5 did not like IT2, they were unlikely to enjoy SW2 either.

- We might observe that the people who rated both IT2 and IT3 tended to give them similar ratings. Thus, we could conclude that user U5 would also give IT3 a low rating, similar to U5's rating of IT2

It is not necessary to predict every blank entry in a utility matrix. Rather, it is only necessary to discover some entries in each row that are likely to be high. In most applications, the recommendation system does not offer users a ranking of all items, but rather suggests a few that the user should value highly. It may not even be necessary to find all items with the highest expected ratings, but only to find a large subset of those with the highest ratings.

2.2 Families of recommender systems

In general, recommendation tools can be categorized into two groups:

- Content-Based systems focus on properties of the items.
- Collaborative-Filtering systems focus on the relationship between users and items.

In this case the artists take the role of items, and users maintain the same role. Since we have no information about artists, except their names, we cannot build a content-based recommender system. Therefore, in the rest of the analysis will only focus on Collaborative-Filtering algorithms.

2.3 Collaborative-Filtering

We will study a member of a broad class of algorithms called latent-factor models. These algorithms try to explain observed interactions between large numbers of users and products through a relatively small number of unobserved, underlying reasons. It is equivalent to explaining why millions of people buy particular albums by describing users and albums in terms of tastes for perhaps tens of genres, tastes which are not directly observable or given as data.

We must formulate the learning problem as a matrix completion problem. Then, use a type of matrix factorization model to "fill in" the missing information. There are implicit ratings users have given certain items and the goal is to predict the ratings for the rest of the items. If there are users and items, there is a matrix which the generic entry represents the rating for item by user. The Matrix has many missing entries indicating unobserved ratings, and the task is to estimate these unobserved ratings.

A popular approach to the matrix completion problem is matrix factorization, where they "summarize" users and items with their latent factors.

Parallel Altenating Least Squares using broadcast variables

This approach takes advantage of the fact that X and Y factor matrices are often very small and can be stored locally on each machine.

- Partition the Ratings RDD by user to create, and similarly partition the Ratings RDD by item to create. There are two copies of the same Ratings RDD, albeit with different partitioning. All ratings by the same user are on the same machine, and in all ratings for same item are on the same machine.
- Broadcast the matrices X and Y. Note these matrices are not RDD of vectors: they are now "local: matrices.

- Using R_1 and Y_1, we can use expressions x_u to compute the update of x_u locally on each machine,
- Using R_2 and X_1, we can use expression y_i from above to compute the update of y_i locally on each machine

A further optimization to this method is to group the and factors matrices into blocks (user blocks and item blocks) and reduce the communication by only sending to each machine the block of users (or items) that are needed to compute the updates at that machine.

This method is called *Block ALS*. This is achieved by precomputing some information about the ratings matrix to determine the "out-links" of each user (which blocks of the items it will contribute to) and "in-link" information for each item (which of the factor vectors it receives from each user block it will depend on). For example, assume machine 1 is responsible for users 1,2,...,37: block 1 of users. The items rated by these users are block 1 of items. Only the factors of block 1 of users and block 1 of items will be broadcasted to machine 1.

Further readings

Other methods for matrix factorization include:

- Low Rank Approximation and Regression in Input Sparsity Time, by Kenneth L. Clarkson, David P. Woodruff. http://arxiv.org/abs/1207.6365
- Generalized Low Rank Models (GLRM), by Madeleine Udell, Corinne Horn, Reza Zadeh, Stephen Boyd. http://arxiv.org/abs/1410.0342
- Matrix Completion and Low-Rank SVD via Fast Alternating Least Squares, by Trevor Hastie, Rahul Mazumder, Jason D. Lee, Reza Zadeh . Statistics Department and ICME, Stanford University, 2014. http://stanford.edu/~rezab/papers/fastals.pdf

3. User Case : Music recommender system

For this user case the data used in the previous sections be used to build a statistical model to recommend artists for users.

3.1 Requirements

According to the properties of data, we need to choose a recommender algorithm that is suitable for this implicit feedback data, this means the algorithm should learn without access to user or artist attributes such as age, genre etc. Therefore, an algorithm of type collaborative filtering is the best choice.

Secondly, the data has some users that have listened to only 1 artist, therefore algorithm that might provide decent recommendations to even these users needs to be utilized. Because at some point, every user starts out with just one play at some point!

Thirdly, an algorithm that is scalable to build large models and create recommendations quickly is required. An algorithm which can run on a distributed system like SPARK or Hadoop is suitable.

Considering these requirements utilizing ALS algorithm in SPARK's MLLIB is a good choice.

3.2 Notes

Currently, MLLIB can only build models from an RDD. This means there are two ways to prepare data:

- Loading into SPARK SQL DataFrame as before, and then access the corresponding RDD by calling <dataframe>.rdd. The invalid data is often successfully dropped by using mode DROPMALFORMED. However, this might not work in all cases. Fortunately, we can use it with this user case.
- Loading data directly to RDD. However, this will require invalid data being managed by the programmer. This is the most reliable way and can work in every case.

The second approach was used for this analysis, it requires more work, but the reward is worth it.

3.3 Cleanup the data

In section Data Integration, misspelled artist IDs have been replaced by the corresponding standard ids by using SPARK SQL API. However, if the data has the invalid entries such that SPARK SQL API is stuck, the best way to work with it is using an RDD.

The goal is:

- Read the input *user_artist_data.txt* and transforms its representation into an output dataset.
- To produce an output "tuple" containing the original user identifier and play counts, with the artist identifier replaced by its most common alias, as found in the *artist_alias.txt dataset*.
- The *artist_alias.txt* file is small so we can use a technique called broadcast variables to make such transformation more efficient.

In [24]:
```
rawArtistAlias = sc.textFile(base + "artist_alias.txt")

def xtractFields(s):
    # Using white space or tab character as separetors,
    # split a line into list of strings
    line = re.split("\s|\t",s,1)
    # if this line has at least 2 characters
    if (len(line) > 1):
        try:
            # try to parse the first and the second components to integer type
            return (int(line[0]), int(line[1]))
        except ValueError:
            # if parsing has any error, return a special tuple
            return (-1,-1)
    else:
        # if this line has less than 2 characters, return a special tuple
        return (-1,-1)

artistAlias = (
                rawArtistAlias
                    # extract fields using function xtractFields
                    .map( lambda x: xtractFields(x))

                    # fileter out the special tuples
                    .filter( lambda x: x != (-1,-1) )
                    # collect result to the driver as a "dictionary"
                    .collectAsMap()
                )
```

In [25]:
```
bArtistAlias = sc.broadcast(artistAlias)
rawUserArtistData = sc.textFile(base + "user_artist_data.txt")

def disambiguate(line):
    [userID, artistID, count] = line.split(' ')
    finalArtistID = bArtistAlias.value.get(artistID,artistID)
    return (userID, finalArtistID,count)


userArtistDataRDD = rawUserArtistData.map(disambiguate)
userArtistDataRDD.take(5)
```

Out[25]:
```
[('1000002', '1', '55'),
 ('1000002', '1000006', '33'),
 ('1000002', '1000007', '8'),
 ('1000002', '1000009', '144'),
 ('1000002', '1000010', '314')]
```

3.4 Training our statistical model

To train a model using ALS, a preference matrix must be used. MLLIB uses the class Rating to support the construction of a distributed preference matrix.

A model can be trained by using ALS.trainImplicit where:

- *training data* is the input data you decide to feed to the ALS algorithm
- *rank* is the number of latten features

We can also use some additional parameters to adjust the quality of the model.

In [26]:
```
from pyspark.mllib.recommendation import ALS, MatrixFactorizationModel, Rating
```

In [27]:
```
allData = userArtistDataRDD.map(lambda r: Rating( float(r[0]),float(r[1]),int(r[
                                                  .repartition(12).cache()
allData.take(5)
```

Out[27]:
```
[Rating(user=1038228, product=1027389, rating=1.0),
 Rating(user=1038228, product=10280656, rating=4.0),
 Rating(user=1038228, product=1028633, rating=2.0),
 Rating(user=1038228, product=1028971, rating=1.0),
 Rating(user=1038228, product=1029494, rating=3.0)]
```

In [28]:
```
#setting parameters
rank=10
iterations=5
lambda_=0.01
alpha=1.0

#training
t0 = time()
model = ALS.trainImplicit(allData, rank)
t1 = time()
print("finish training model in %f secs" % (t1 - t0))
```

```
finish training model in 91.764442 secs
```

In [31]:
```
#! hdfs dfs -rm -R -f -skipTrash lastfm_model.spark
#print('Delete old model')
model.save(sc,'music_rec_model.spark')
print('Save new model')
```

```
Save new model
```

In [32]:
```
t0 = time()
model = MatrixFactorizationModel.load(sc, 'music_rec_model.spark')
t1 = time()
print("finish loading model in %f secs" % (t1 - t0))
```

```
finish loading model in 1.624458 secs
```

In [33]:
```
model.userFeatures().first()
```

Out[33]:
```
(90,
 array('d', [-0.05403498187661171, -0.00984848104417324, -0.0005837124772369862,
0.003350968472659588, -0.06338433176279068, -0.035387467592954636, 0.072135150432
173, -0.03624201565980911]))
```

In [34]:
```
# Make five reccommendations to user 2093760
recommendations = (model.recommendProducts(2093760,5))
print(recommendations)
# construct set of recommendated artists
recArtist = set(rating[1] for rating in recommendations)
recArtist
```

```
[Rating(user=2093760, product=1007614, rating=0.030772450232132396), Rating(user=2093760, pr
rating=0.029059107013603248), Rating(user=2093760, product=2814, rating=0.02767556526645912)
r=2093760, product=829, rating=0.0275336505441955), Rating(user=2093760, product=1037970, ra
8925288733955)]
```

Out[34]:
```
{829, 2814, 4605, 1007614, 1037970}
```

In [35]:
```
# construct data of artists (artist_id, artist_name)

rawArtistData = sc.textFile(base + "artist_data.txt")

def xtractFields(s):
    line = re.split("\s|\t",s,1)
    if (len(line) > 1):
        try:
            return (int(line[0]), str(line[1].strip()))
        except ValueError:
            return (-1,"")
    else:
        return (-1,"")

artistByID = rawArtistData.map(xtractFields).filter(lambda x: x[0] > 0)
```

In [36]:
```
# Filter in those artists, get just artist, and print
def artistNames(line):
#     [artistID, name]
    if (line[0] in recArtist):
        return True
    else:
        return False

recList = artistByID.filter(artistNames).values().collect()

print(recList)
```

```
['50 Cent', 'Snoop Dogg', 'Nas', 'Jay-Z', 'Kanye West']
```

In [37]:
```
def unpersist(model):
    model.userFeatures().unpersist()
    model.productFeatures().unpersist()

# uncache data and model when they are no longer used
unpersist(model)
```

3.5 Evaluating Recommendation Quality

In this section, we study how to evaluate the quality of our model. It's hard to say how good the recommendations are. One of several methods approach to evaluate a recommender based on its ability to rank good items (artists) high in a list of recommendations. The problem is how to define "good artists". Currently, by training all data, "good artists" is defined as "artists the user has listened to", and the recommender system has already received all of this information as input. It could

trivially return the users previously-listened artists as top recommendations and score perfectly. Indeed, this is not useful, because the recommender's is used to recommend artists that the user has never listened to.

To overcome that problem, we can hide the some of the artist play data and only use the rest to train model. Then, this held-out data can be interpreted as a collection of "good" recommendations for each user. The recommender is asked to rank all items in the model, and the rank of the held-out artists are examined. Ideally the recommender places all of them at or near the top of the list.

The recommender's score can then be computed by comparing all held-out artists' ranks to the rest. The fraction of pairs where the held-out artist is ranked higher is its score. 1.0 is perfect, 0.0 is the worst possible score, and 0.5 is the expected value achieved from randomly ranking artists.

AUC(Area Under the Curve) can be used as a metric to evaluate model. It is also viewed as the probability that a randomly-chosen "good" artist ranks above a randomly-chosen "bad" artist.

Next, we split the training data into 2 parts: trainData and cvData with ratio 0.7:0.3 respectively, where trainData is the dataset that will be used to train model. Then we write a function to calculate AUC to evaluate the quality of our model.

Split the data into trainData and cvData with ratio 0.9:0.1 and use the first part to train a statistic model with:

- rank=10
- iterations=5
- lambda_=0.01
- alpha=1.0

```
In [38]: trainData, cvData = allData.randomSplit([0.7,0.3],1)
         trainData.cache()
         cvData.cache()
```

```
Out[38]: PythonRDD[332] at RDD at PythonRDD.scala:49
```


```
In [38]: trainData, cvData = allData.randomSplit([0.7,0.3],1)
         trainData.cache()
         cvData.cache()
```

```
Out[38]: PythonRDD[332] at RDD at PythonRDD.scala:49
```

In [39]:
```
#training
t0 = time()
model = ALS.trainImplicit(ratings=trainData,rank=rank,iterations=iterations,lamb
a)
t1 = time()
print("finish training model in %f secs" % (t1 - t0))
```

```
finish training model in 74.359045 secs
```

In [40]:
```
# Get all unique artistId, and broadcast them
allItemIDs = np.array(allData.map(lambda x: x[1]).distinct().collect())
bAllItemIDs = sc.broadcast(allItemIDs)
```

In [ ]:
```
from random import randint

# Depend on the number of item in userIDAndPosItemIDs,
# create a set of "negative" products for each user. These are randomly chosen
# from among all of the other items, excluding those that are "positive" for the
# NOTE 1: mapPartitions operates on many (user,positive-items) pairs at once
# NOTE 2: flatMap breaks the collections above down into one big set of tuples
def xtractNegative(userIDAndPosItemIDs):
    def pickEnoughNegatives(line):
        userID = line[0]
        posItemIDSet = set(line[1])
        #posItemIDSet = line[1]
        negative = []
        allItemIDs = bAllItemIDs.value
        # Keep about as many negative examples per user as positive. Duplicates
        i = 0
        while (i < len(allItemIDs) and len(negative) < len(posItemIDSet)):
            itemID = allItemIDs[randint(0,len(allItemIDs)-1)]
            if itemID not in posItemIDSet:
                negative.append(itemID)
            i += 1

        # Result is a collection of (user,negative-item) tuples
        return map(lambda itemID: (userID, itemID), negative)

    # Init an RNG and the item IDs set once for partition
    # allItemIDs = bAllItemIDs.value
    return map(pickEnoughNegatives, userIDAndPosItemIDs)
```

In [ ]:
```
def ratioOfCorrectRanks(positiveRatings, negativeRatings):

    # find number elements in arr that has index >= start and has value smaller t
    # arr is a sorted array
    def findNumElementsSmallerThan(arr, x, start=0):
        left = start
        right = len(arr) -1
        # if x is bigger than the biggest element in arr
        if start > right or x > arr[right]:
            return right + 1
        mid = -1
        while left <= right:
            mid = (left + right) // 2
            if arr[mid] < x:
                left = mid + 1
            elif arr[mid] > x:
                right = mid - 1
            else:
                while mid-1 >= start and arr[mid-1] == x:
                    mid -= 1
                return mid
        return mid if arr[mid] > x else mid + 1
```

In [ ]:
```
    ## AUC may be viewed as the probability that a random positive item scores
    ## higher than a random negative one. Here the proportion of all positive-n
    ## pairs that are correctly ranked is computed. The result is equal to the
    correct = 0 ## L
    total = 0 ## L

    # sorting positiveRatings array needs more cost
    #positiveRatings = np.array(map(lambda x: x.rating, positiveRatings))

    negativeRatings = list(map(lambda x:x.rating, negativeRatings))

    #np.sort(positiveRatings)
    negativeRatings.sort()# = np.sort(negativeRatings)
    total = len(positiveRatings)*len(negativeRatings)

    for positive in positiveRatings:
        # Count the correctly-ranked pairs
        correct += findNumElementsSmallerThan(negativeRatings, positive.rating)

    ## Return AUC: fraction of pairs ranked correctly
    return float(correct) / total
```

In [ ]:
```
def calculateAUC(positiveData, bAllItemIDs, predictFunction):
    # Take held-out data as the "positive", and map to tuples
    positiveUserProducts = positiveData.map(lambda r: (r[0], r[1]))
    # Make predictions for each of them, including a numeric score, and gathe
    positivePredictions = predictFunction(positiveUserProducts).groupBy(lambda

    # Create a set of "negative" products for each user. These are randomly c
    # from among all of the other items, excluding those that are "positive"
    negativeUserProducts = positiveUserProducts.groupByKey().mapPartitions(xt
    # Make predictions on the rest
    negativePredictions = predictFunction(negativeUserProducts).groupBy(lambda

    return (
            positivePredictions.join(negativePredictions)
                .values()
                .map(
                    lambda positive_negativeRatings: ratioOfCorrectRanks(posit
                )
                .mean()
            )
```

In [42]:
```
t0 = time()
auc = calculateAUC( cvData,bAllItemIDs, model.predictAll)
t1 = time()
print("auc=",auc)
print("finish in %f seconds" % (t1 - t0))
```

```
auc= 0.96070668941573
finish in 79.103230 seconds
```

In [43]:
```
bListenCount = sc.broadcast(trainData.map(lambda r: (r[1], r[2])).reduceByKey(la
y).collectAsMap())
def predictMostListened(allData):
    return allData.map(lambda r: Rating(r[0], r[1], bListenCount.value.get( r[1]
```

In [44]:
```
auc = calculateAUC(cvData,bListenCount, predictMostListened)
print("AUC score:" + str(auc))
```

```
AUC score:0.9361065418733973
```

3.6 Personalized recommendations with ALS: Hyperparameters tuning

In the previous section, we build our models with some given paramters without any knowledge about them. Actually, choosing the best parameters' values is very important. It can significantly affect the quality of models. Especially, with the current implementation of ALS in MLLIB, these parameters are not learned by the algorithm,

and must be chosen by the caller. The following parameters should get consideration before training models:

- rank = 10: the number of latent factors in the model, or equivalently, the number of columns in the user-feature and product-feature matrices. In non-trivial cases, this is also their rank.

- iterations = 5: the number of iterations that the factorization runs. Instead of runing the algorithm until RMSE converged which actually takes very long time to finish with large datasets, we only let it run in a given number of iterations. More iterations take more time but may produce a better factorization.

- lambda_ = 0.01: a standard overfitting parameter. Higher values resist overfitting, but values that are too high hurt the factorization's accuracy.

- alpha = 1.0: controls the relative weight of observed versus unobserved userproduct interactions in the factorization.

Although all of them have impact on the models' quality, iterations is more of a constraint on resources used in the factorization. So, rank, lambda_ and alpha can be considered hyperparameters to the model. We will try to find "good" values for them. Indeed, the values of hyperparameter are not necessarily optimal. Choosing good hyperparameter values is a common problem in machine learning. The most basic way to choose values is to simply try combinations of values and evaluate a metric for each of them, and choose the combination that produces the best value of the metric.

In [65]:
```
evaluations = []

for rank in [10, 50]:
    for lambda_ in [1.0, 0.0001]:
        for alpha in [1.0, 40.0]:
            print("Train model with rank=%d lambda_=%f alpha=%f" % (rank, lambda_
            # with each combination of params, we should run multiple times and g
            # for simple, we only run one time.
            model = ALS.trainImplicit(ratings=trainData,rank=rank,iterations=5,la
a_ ,alpha=alpha)
            auc = calculateAUC(cvData,bListenCount,model.predictAll)

            evaluations.append(((rank, lambda_, alpha), auc))

            unpersist(model)

evaluations.sort(key = lambda x: -x[1])
evalDataFrame = pd.DataFrame(data=evaluations)
print(evalDataFrame)

trainData.unpersist()
cvData.unpersist()
```

```
Train model with rank=10 lambda_=1.000000 alpha=1.000000
Train model with rank=10 lambda_=1.000000 alpha=40.000000
Train model with rank=10 lambda_=0.000100 alpha=1.000000
Train model with rank=10 lambda_=0.000100 alpha=40.000000
Train model with rank=50 lambda_=1.000000 alpha=1.000000
Train model with rank=50 lambda_=1.000000 alpha=40.000000
Train model with rank=50 lambda_=0.000100 alpha=1.000000
Train model with rank=50 lambda_=0.000100 alpha=40.000000
                      0         1
0     (10, 1.0, 40.0)  0.973829
1   (10, 0.0001, 40.0)  0.971861
2     (50, 1.0, 40.0)  0.971527
3   (50, 0.0001, 40.0)  0.970065
4      (10, 1.0, 1.0)  0.964493
5      (50, 1.0, 1.0)  0.959280
6    (10, 0.0001, 1.0)  0.958409
7    (50, 0.0001, 1.0)  0.942701
```

Out[65]: PythonRDD[349] at RDD at PythonRDD.scala:49

The combination of parameters that gets the highest AUC is: rank = 10 ; lambda = 1.0 ; alpha = 40

In [45]:
```
model = ALS.trainImplicit(ratings=trainData, rank=10, iterations=5, lambda_=1.0,
allData.unpersist()

userID = 2093760
recommendations = model.recommendProducts(userID,5)

recArtist = set(rating[1] for rating in recommendations)

# Filter in those artists, get just artist, and print
def artistNames(line):
#     [artistID, name]
    if (line[0] in recArtist):
        return True
    else:
        return False

recList = artistByID.filter(artistNames).values().collect()
print(recList)

unpersist(model)
```

```
['Kent', 'Oasis', 'The Killers', 'Kaiser Chiefs', 'Unknown']
```

It seems that the result of top 5 recommended artists does not change when we add more parameters to the model or modify them (such as lambda, alpha). I think that we should extend to top 50 or top 70 to see how the recommendation changing. Because I think that lambda and alpha parameters affect only when the rating of each artist approximate the mean of total ratings for one user (Top 5 artists have the much larger rating value than others, top 5 artists do not change with modified parameter)

This raises me the question that how strong the impact of those parameters? Is it really nessessary to put into our model when we just need to retrieve a small number of top artists (let's say, less than top 10)?

Additional work

1/ Changing the proportion of splitting data

In this experiment, I will change the percentage of training data to 50%, 80%, 90% and 99% respectively. The purpose is to observe the changing of AUC score

In [72]:
```
trainData, cvData = allData.randomSplit([0.5,0.5],1)
trainData.cache()
cvData.cache()

t0 = time()
model = ALS.trainImplicit(ratings=trainData,rank=rank,iterations=iterations,lamb
,alpha=alpha)
t1 = time()
print("finish training model in %f secs" % (t1 - t0))

t0 = time()
auc = calculateAUC( cvData,bAllItemIDs, model.predictAll)
t1 = time()
print("auc=",auc)
print("finish in %f seconds" % (t1 - t0))
```

```
finish training model in 242.592275 secs
auc= 0.9721443801890255
finish in 130.203251 seconds
```

In [73]:
```
trainData, cvData = allData.randomSplit([0.8,0.2],1)
trainData.cache()
cvData.cache()

t0 = time()
model = ALS.trainImplicit(ratings=trainData,rank=rank,iterations=iterations,lamb
,alpha=alpha)
t1 = time()
print("finish training model in %f secs" % (t1 - t0))

t0 = time()
auc = calculateAUC( cvData,bAllItemIDs, model.predictAll)
t1 = time()
print("auc=",auc)
print("finish in %f seconds" % (t1 - t0))
```

```
finish training model in 338.598379 secs
auc= 0.9808745215544642
finish in 61.614985 seconds
```

In [74]:
```
trainData, cvData = allData.randomSplit([0.9,0.1],1)
trainData.cache()
cvData.cache()

t0 = time()
model = ALS.trainImplicit(ratings=trainData,rank=rank,iterations=iterations,lamb
,alpha=alpha)
t1 = time()
print("finish training model in %f secs" % (t1 - t0))

t0 = time()
auc = calculateAUC( cvData,bAllItemIDs, model.predictAll)
t1 = time()
print("auc=",auc)
print("finish in %f seconds" % (t1 - t0))
```

```
finish training model in 247.797857 secs
auc= 0.9847853981146301
finish in 39.068620 seconds
```

In [75]:
```
trainData, cvData = allData.randomSplit([0.99,0.01],1)
trainData.cache()
cvData.cache()

t0 = time()
model = ALS.trainImplicit(ratings=trainData,rank=rank,iterations=iterations,lamb
,alpha=alpha)
t1 = time()
print("finish training model in %f secs" % (t1 - t0))

t0 = time()
auc = calculateAUC( cvData,bAllItemIDs, model.predictAll)
t1 = time()
print("auc=",auc)
print("finish in %f seconds" % (t1 - t0))
```

```
finish training model in 325.923175 secs
auc= 0.9856858729938448
finish in 16.474116 seconds
```

This is obviouly that the proportion of splitting data does affect the result of the AUC score. The more training data, the higher AUC score. In my opinion, having higher AUC score does not mean that the model is working well in general. Because high percentage of training data may prone to overfitting.

2/ Expand the the list of recommended artist

As my hypothesis, in this experiment, instead of retrieving top 5 artist, I try to retrieve top 50 and 70 artist with modified parameters to see the differences the result compare to the model with standard parameters

Model with standard parameters (lambda and alpha are set by default)

In [76]:
```
trainData, cvData = allData.randomSplit([0.7,0.3],1)
trainData.cache()
cvData.cache()
```

Out[76]:
```
PythonRDD[3102] at RDD at PythonRDD.scala:49
```

In [77]:
```
model50 = ALS.trainImplicit(ratings=trainData, rank=10, iterations=5)
trainData.unpersist()

userID = 2093760
recommendations50 = model50.recommendProducts(userID,50)

recArtist = set(rating[1] for rating in recommendations50)

recList50 = artistByID.filter(artistNames).values().collect()

print(recList50)

unpersist(model)
```

```
['Eric Clapton', 'Notorious B.I.G.', '50 Cent', 'Sublime', 'Snoop Dogg', 'RJD2',
Service', 'De La Soul', 'Nas', 'Jay-Z', 'Zero 7', 'The Chemical Brothers', 'Kanye
E*R*D', 'Wu-Tang Clan', 'Daft Punk', 'Mos Def', 'Cake', 'Bob Marley', 'Dr. Dre',
'Rage Against the Machine', '2Pac', 'Jack Johnson', 'The Beatles', 'Eminem', 'Mod
'Pearl Jam', 'Thievery Corporation', 'A Tribe Called Quest', 'Jimi Hendrix', 'Led
'Radiohead', 'Red Hot Chili Peppers', 'Coldplay', 'Dave Matthews Band', 'U2', 'Go
cubus', 'Miles Davis', 'DJ Shadow', 'Pink Floyd', 'Jurassic 5', 'Beck', 'Outkast'
', 'The Game', 'Talib Kweli', '311', 'Beastie Boys']
```

In [78]:
```
model50_1 = ALS.trainImplicit(ratings=trainData, rank=10, iterations=5, lambda_=
ha= 40.0)
trainData.unpersist()

userID = 2093760
recommendations50_1 = model50_1.recommendProducts(userID,50)

recArtist = set(rating[1] for rating in recommendations50_1)

recList50_1 = artistByID.filter(artistNames).values().collect()

print(recList50_1)

unpersist(model)
```

```
['Bloc Party', 'Snow Patrol', '[unknown]', 'The Postal Service', 'Death Cab for C
e', 'Damien Rice', 'Bob Dylan', 'Johnny Cash', 'Elliott Smith', 'The Who', 'Cake'
', 'Bruce Springsteen', 'The Shins', 'Jack Johnson', 'Jeff Buckley', 'Bright Eyes
h', 'Pixies', 'The Smiths', 'The Beatles', 'Modest Mouse', 'The Strokes', 'The Wh
', 'R.E.M.', 'Franz Ferdinand', 'Simon & Garfunkel', 'The Cure', 'Led Zeppelin',
'David Bowie', 'The Mars Volta', 'The Killers', 'Moby', 'Coldplay', 'U2', 'Blur',
Boys', 'Weezer', 'Interpol', 'Pink Floyd', 'The Rolling Stones', 'Beck', 'Kaiser
ey Might Be Giants', 'Ben Folds', 'Beastie Boys', 'Björk', 'Hot Hot Heat']
```

In [79]:
```
temp3 = [item for item in recList50 if item not in recList50_1]
print(temp3)
print("\n Number of different artists between standard model and model_1:" + str
3)))
```

```
['Eric Clapton', 'Notorious B.I.G.', '50 Cent', 'Sublime', 'Snoop Dogg', 'RJD2',
', 'Nas', 'Jay-Z', 'Zero 7', 'The Chemical Brothers', 'Kanye West', 'N*E*R*D', 'W
', 'Daft Punk', 'Mos Def', 'Bob Marley', 'Dr. Dre', 'Ludacris', 'Rage Against the
2Pac', 'Eminem', 'Pearl Jam', 'Thievery Corporation', 'A Tribe Called Quest', 'Ji
'Red Hot Chili Peppers', 'Dave Matthews Band', 'Gorillaz', 'Incubus', 'Miles Davi
ow', 'Jurassic 5', 'Outkast', 'The Roots', 'The Game', 'Talib Kweli', '311']

 Number of different artists between standard model and model_1:38
```

In [80]:
```
model50_2 = ALS.trainImplicit(ratings=trainData, rank=10, iterations=5, lambda_=
1.0)
trainData.unpersist()

userID = 2093760
recommendations50_2 = model50_2.recommendProducts(userID,50)

recArtist = set(rating[1] for rating in recommendations50_2)

recList50_2 = artistByID.filter(artistNames).values().collect()

print(recList50_2)

unpersist(model)
```

```
['Notorious B.I.G.', 'Fabolous', '50 Cent', 'Mobb Deep', 'DMX', 'Snoop Dogg', 'Us
', 'Busta Rhymes', 'De La Soul', 'Nas', 'Ja Rule', 'Jay-Z', 'Will Smith', 'Black
'Gang Starr', 'Kanye West', 'N*E*R*D', 'D12', 'Wu-Tang Clan', 'Mos Def', 'Bob Mar
re', 'Ludacris', '2Pac', 'Eminem', 'A Tribe Called Quest', 'Bob Marley & the Wail
sic 5', 'Jedi Mind Tricks', 'Method Man', 'Outkast', 'The Roots', 'Lil Jon & The
yz', 'Xzibit', 'G-Unit', 'The Game', 'Lloyd Banks', 'Jay-Z and Linkin Park', 'Bon
rmony', 'Talib Kweli', 'Obie Trice', 'Chingy', 'Atmosphere', 'Dilated Peoples', '
sta', 'Cypress Hill', 'Ice Cube', 'Common']
```

In [81]:
```
temp4 = [item for item in recList50 if item not in recList50_2]
print(temp4)
print("\n Number of different artists between standard model and model_2:" + str
4)))
```

```
['Eric Clapton', 'Sublime', 'RJD2', 'The Postal Service', 'Zero 7', 'The Chemical
'Daft Punk', 'Cake', 'Rage Against the Machine', 'Jack Johnson', 'The Beatles', '
', 'Pearl Jam', 'Thievery Corporation', 'Jimi Hendrix', 'Led Zeppelin', 'Radiohea
Chili Peppers', 'Coldplay', 'Dave Matthews Band', 'U2', 'Gorillaz', 'Incubus', 'M
'DJ Shadow', 'Pink Floyd', 'Beck', '311', 'Beastie Boys']

 Number of different artists between standard model and model_2:29
```

Conclusion: As the result has shown above, I dont need to expand the result list to top 70 artists anymore. It is clearly proving my hypothesis is true

How we configure the lambda and alpha makes a huge impact on the retrieved result: In my example, the top artists that highly recommended to user="2093760" is ['50Cent', 'Snoopdog', Notorious B.I.G] (they appear in both standard model and modified models). Others artists may vary from different models.

We may not need to include lambda and alpha to the model if we only retreive the top result less than 10 artists.

Proposed method:

In my opinion: We should cut out outliers (top 5 artists). With this method, recommendation will be more diverse and accurate.

Summary

In this notebook, we introduce an algorithm to do matrix factorization and the way of using it to make recommendation. Further more, we studied how to build a large-scale recommender system on SPARK using ALS algorithm and evaluate its quality. Finally, a simple approach to choose good parameters is mentioned.