

Task 1. Building a music recommender system

As its name implies, a recommender system is a tool that helps predicting what a user may or may not like among a list of given items. In some sense, you can view this as an alternative to content search, as recommendation engines help users discover products or content that they may not come across otherwise. For example, Facebook suggests friends and pages to users. Youtube recommends videos which users may be interested in. Amazon suggests the products which users may need... Recommendation engines engage users to services, can be seen as a revenue optimization process, and in general help maintaining interest in a service.

In this notebook, we study how to build a simple recommender system: we focus on music recommendations, and we use a simple algorithm to predict which items users might like, that is called ALS, alternating least squares.

Goals

In this lecture, we expect students to:

- Revisit (or learn) recommender algorithms
- Understand the idea of Matrix Factorization and the ALS algorithm (serial and parallel versions)
- Build a simple model for a real usecase: music recommender system
- Understand how to validate the results

Steps

We assume students to work outside lab hours on the learning material. These are the steps by which we guide students, during labs, to build a good basis for the end-to-end development of a recommender system:

- Inspect the data using Spark SQL, and build some basic, but very valuable knowledge about the information we have at hand
- Formally define what is a sensible algorithm to achieve our goal: given the "history" of user taste for music, recommend new music to discover. Essentially, we want to build a statistical model of user preferences such that we can use it to "predict" which additional music the user could like
- With our formal definition at hand, we will learn different ways to implement such an algorithm. Our goal here is to illustrate what are the difficulties to overcome when implementing a (parallel) algorithm
- Finally, we will focus on an existing implementation, available in the Apache Spark MLlib, which we will use out of the box to build a reliable statistical model

Now, you may think at this point we will be done!

Well, you'd better think twice: one important topic we will cover in all our Notebooks is **how to validate the results we obtain**, and **how to choose good parameters to train models** especially when using an "opaque" library for doing the job. As a consequence, we will focus on the statistical validation of our recommender system.

Important note for grading

This notebook displays a series of questions, that we use to grade the work done. Since the following questions are rather basic, and do not change much from year to year, they only allow reaching the grade 10/20. Additional points can be gained by showing **originality, depth, algorithmic design and implementations** beyond that used in the notebook. Remember that this should become your own notebook: there is ample room for creativity!

1. Data

Understanding data is one of the most important part when designing any machine learning algorithm. In this notebook, we will use a data set published by Audioscrobbler - a music recommendation system for last.fm. Audioscrobbler is also one of the first internet streaming radio sites, founded in 2002. It provided an open API for "scrobbling", or recording listeners' plays of artists' songs. last.fm used this information to build a powerful music recommender engine.

1.1. Data schema

Unlike a rating dataset which contains information about users' preference for products (one star, 3 stars, and so on), the datasets from Audioscrobbler only has information about events: specifically, it keeps track of how many times a user played songs of a given artist and the names of artists. That means it carries less information than a rating: in the literature, this is called explicit vs. implicit ratings.

Reading material

- [Implicit Feedback for Inferring User Preference: A Bibliography](http://people.csail.mit.edu/teevan/work/publications/papers/sigir-forum03.pdf)
(<http://people.csail.mit.edu/teevan/work/publications/papers/sigir-forum03.pdf>)
- [Comparing explicit and implicit feedback techniques for web retrieval: TREC-10 interactive track report](http://trec.nist.gov/pubs/trec10/papers/glasgow.pdf)
(<http://trec.nist.gov/pubs/trec10/papers/glasgow.pdf>)
- [Probabilistic Models for Data Combination in Recommender Systems](http://mlg.eng.cam.ac.uk/pub/pdf/WilGha08.pdf)
(<http://mlg.eng.cam.ac.uk/pub/pdf/WilGha08.pdf>)

The data we use in this Notebook is available in 3 files (these files are stored in our HDFS layer, in the directory /datasets/lastfm):

- **user_artist_data.txt**: It contains about 140,000+ unique users, and 1.6 million unique artists. About 24.2 million users' plays of artists' are recorded, along with their count. It has 3 columns separated by spaces:

UserID	ArtistID	PlayCount
...

- **artist_data.txt** : It provides the names of each artist by their IDs. It has 2 columns separated by tab characters (**\t**).

ArtistID	Name
...	...

- **artist_alias.txt**: Note that when plays are scrobbled, the client application submits the name of the artist being played. This name could be misspelled or nonstandard. For example, "The Smiths", "Smiths, The", and "the smiths" may appear as distinct artist IDs in the data set, even though they are plainly the same. **artist_alias.txt** maps artist IDs that are known misspellings or variants to the canonical ID of that artist. The data in this file has 2 columns separated by tab characters (**\t**).

MisspelledArtistID	StandardArtistID
...	...

1.2. Understanding data: simple descriptive statistic

In order to choose or design a suitable algorithm for achieving our goals, given the data we have, we should first understand data characteristics. To start, we import the necessary packages to work with regular expressions, Data Frames, and other nice features of our programming environment.

```
In [1]: 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 = "/datasets/lastfm/"
```

[Group 6]

magic function %matplotlib inline : plot inline in Notebook

```
In [1]: !hdfs dfs -ls /datasets/lastfm
!hdfs dfs -cat /datasets/lastfm/artist_data.txt | head -10
```

```
Found 4 items
-rw-r--r--  3 zoeadmin supergroup      1273 2016-02-17 18:09 /datasets/last
fm/README.txt
-rw-r--r--  3 zoeadmin supergroup    2932731 2016-02-17 18:09 /datasets/last
fm/artist_alias.txt
-rw-r--r--  3 zoeadmin supergroup    55963575 2016-02-17 18:09 /datasets/last
fm/artist_data.txt
-rw-r--r--  3 zoeadmin supergroup    426761761 2016-02-17 18:09 /datasets/last
fm/user_artist_data.txt
1134999 06Crazy Life
6821360 Pang Nakarin
10113088      Terfel, Bartoli- Mozart: Don
10151459      The Flaming Sidebur
6826647 Bodenstandig 3000
10186265      Jota Quest e Ivete Sangalo
6828986 Toto_XX (1977
10236364      U.S Bombs -
1135000 artist formally know as Mat
10299728      Kassierer - Musik für beide Ohren
cat: Unable to write to output stream.
```

Question 1

Question 1.0 (Non-grading)

Using SPARK SQL, load data from /datasets/lastfm/user_artist_data.txt and show the first 20 entries (via function show()).

For this Notebook, from a programming point of view, we are given the schema for the data we use, which is as follows:

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

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

```
In [2]: 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 is accessed.
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
```

[Group 6]

pyspark.sql.StructType: The data type representing rows. A StructType object comprises a list of **StructFields**.

Question 1.1:

How many distinct users do we have in our data?

```
In [3]: uniqueUsers = userArtistDF.select('userID').distinct().count()
print (type(userArtistDF.select('userID')))
#uniqueUsers = userArtistDF.select('userID').groupby(userArtistDF['userID']).count().count()

print("Total n. of users: ", uniqueUsers)

<class 'pyspark.sql.dataframe.DataFrame'>
Total n. of users: 148111
```

Question 1.2

How many distinct artists do we have in our data ?

```
In [4]: uniqueArtists = userArtistDF.select('artistID').distinct().count()
print("Total n. of artists: ", uniqueArtists)

Total n. of artists: 1631028
```

Question 1.3

One limitation of Spark MLlib's ALS implementation - which we will use later - is that it requires IDs for users and items to be nonnegative 32-bit integers. This means that IDs larger than Integer.MAX_VALUE, or 2147483647, can't be used. So we need to check whether this data set conforms to the strict requirements of our library.

What are the maximum and minimum values of column userID ?

HINT: Refer to section 4.3 of Laboratory 2.

```
In [5]: #userArtistDF....show()
minimum=userArtistDF.select('userID').orderBy('userID', ascending=1).take(1)
maximum=userArtistDF.select('userID').orderBy('userID',ascending=0).take(1)
print (minimum)
print (maximum)

[Row(userID=90)]
[Row(userID=2443548)]
```

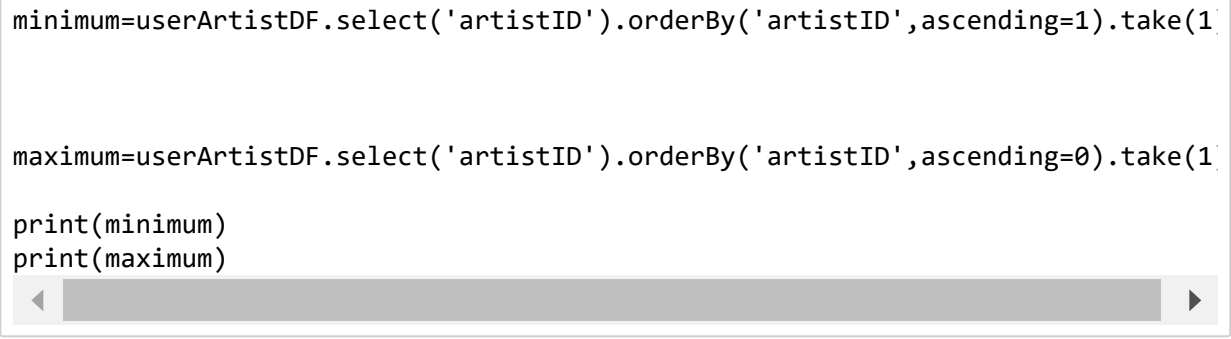
Question 1.4

What is the maximum and minimum values of column artistID ?

```
In [6]: minimum=userArtistDF.select('artistID').orderBy('artistID',ascending=1).take(1)

maximum=userArtistDF.select('artistID').orderBy('artistID',ascending=0).take(1)

print(minimum)
print(maximum)
```



```
[Row(artistID=1)]
[Row(artistID=10794401)]
```

We just discovered that we have a total of 148,111 users in our dataset. Similarly, we have a total of 1,631,028 artists in our 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.

One thing we can see here is that SPARK SQL provides very concise and powerful methods for data analytics (compared to using RDD and their low-level API). You can see more examples [here](https://databricks.com/blog/2015/06/02/statistical-and-mathematical-functions-with-dataframes-in-spark.html) (<https://databricks.com/blog/2015/06/02/statistical-and-mathematical-functions-with-dataframes-in-spark.html>).

Next, we might want to understand better user activity and artist popularity.

Here is a list of simple descriptive queries that helps us reaching these purposes:

- How many times each user has played a song? This is a good indicator of who are the **most active users** of our service. Note that a very active user with many play counts does not necessarily mean that the user is also "curious"! Indeed, she could have played the same song several times.
- How many play counts for each artist? This is a good indicator of the **artist popularity**. Since we do not have time information associated to our data, we can only build a, e.g., top-10 ranking of the most popular artists in the dataset. Later in the notebook, we will learn that our dataset has a very "loose" definition about artists: very often artist IDs point to song titles as well. This means we have to be careful when establishing popular artists. Indeed, artists whose data is "well formed" will have the correct number of play counts associated to them. Instead, artists that appear mixed with song titles may see their play counts "diluted" across their songs.

Question 2

Question 2.1

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


```
In [7]: # Compute user activity
# We are interested in how many playcounts each user has scored.
userArtistDF.groupBy('userID').sum('playCount').orderBy('sum(playCount)', ascending=1).show()
userActivity = userArtistDF.groupBy('userID').sum('playCount').collect()

print(userActivity[0:5])
```

```
+-----+-----+
| userID|sum(playCount)|
+-----+-----+
|2361719|              1|
|1000866|              1|
|2174815|              1|
|2439629|              1|
|2019325|              1|
|2426901|              1|
|2026801|              1|
|1051273|              1|
|2334525|              1|
|2440776|              1|
|2164247|              1|
|2329216|              1|
|2211957|              1|
|2357159|              1|
|2167896|              1|
|2427475|              1|
|2213063|              1|
|2438308|              1|
|2232063|              1|
|2438838|              1|
+-----+-----+
only showing top 20 rows
```

```
[Row(userID=1041783, sum(playCount)=8730), Row(userID=1042771, sum(playCount)=3), Row(userID=1043190, sum(playCount)=8), Row(userID=1043621, sum(playCount)=125), Row(userID=1043703, sum(playCount)=2609)]
```

Question 2.2

Plot CDF (or ECDF) of the number of play counts per User ID.

Explain and comment the figure you just created:

- for example, look at important percentiles (25%, median, 75%, tails such as >90%) and cross check with what you have found above to figure out if the result is plausible.
- discuss about your users, with respect to the application domain we target in the notebook: you will notice that for some users, there is very little interaction with the system, which means that maybe recommending something to them is going to be more difficult than for other users who interact more with the system.
- look at outliers and reason about their impact on your recommender algorithm

```
In [5]: pdf = pd.DataFrame(data=userActivity)
Y = np.sort(pdf[1])
yvals=np.arange(len(Y))/float(len(Y))
#print("PDF:",pdf)
#print("pdf[1]:\n",pdf[1])
#print("Y:\n",Y)
print("np.arange(len(Y)):\n",np.arange(len(Y)))
plt.plot(Y,yvals)
plt.xlabel('Play Counts')
plt.ylabel('ECDF')
plt.grid(True,which="both",ls="-")
plt.title('Fig 1 ECDF of number of play counts per User ID')
plt.show()

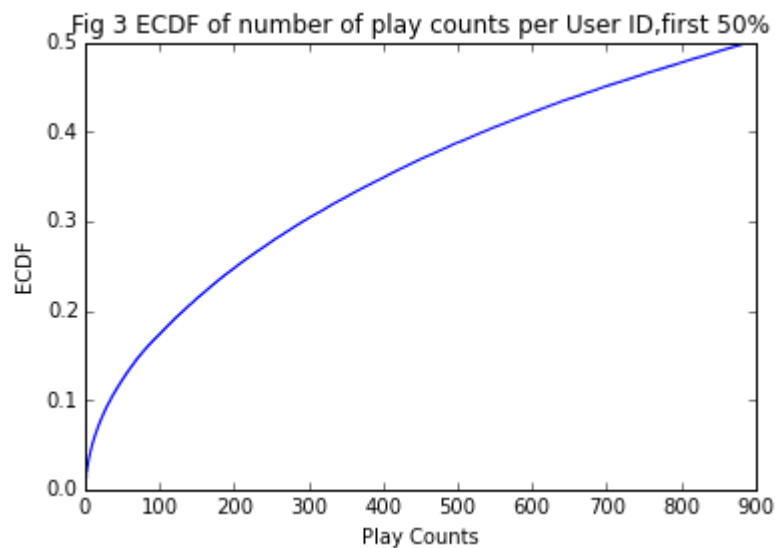
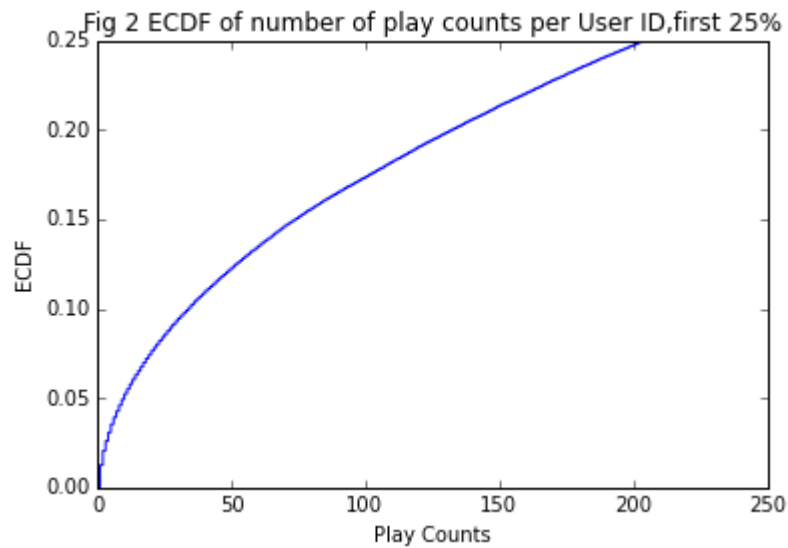
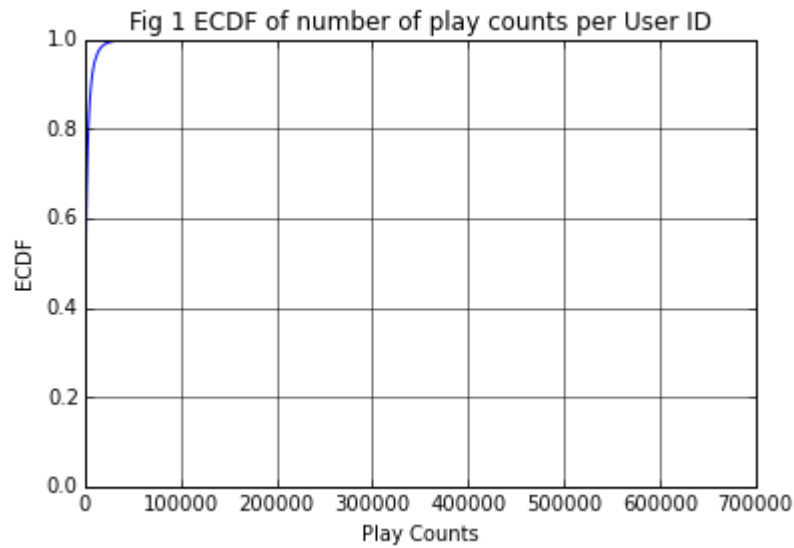
#Focus on first 25% users
plt.plot(Y[0:37000],yvals[0:37000])
plt.xlabel('Play Counts')
plt.ylabel('ECDF')
plt.title('Fig 2 ECDF of number of play counts per User ID,first 25%')
plt.show()

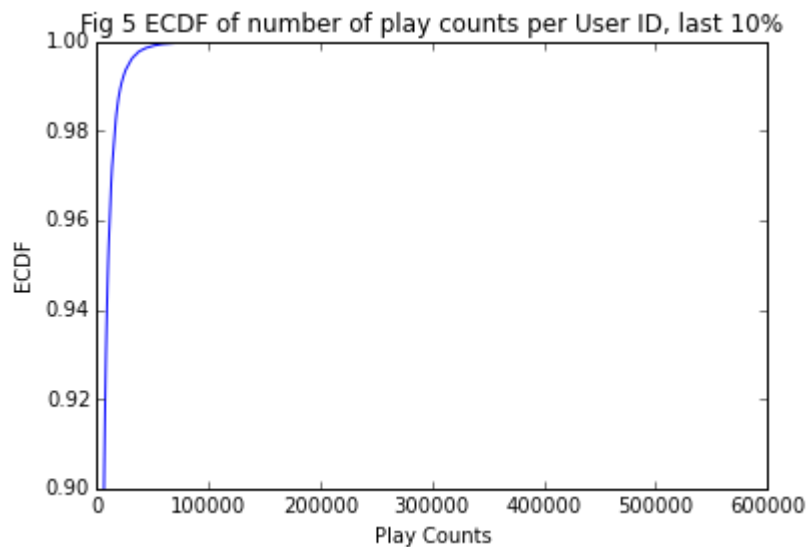
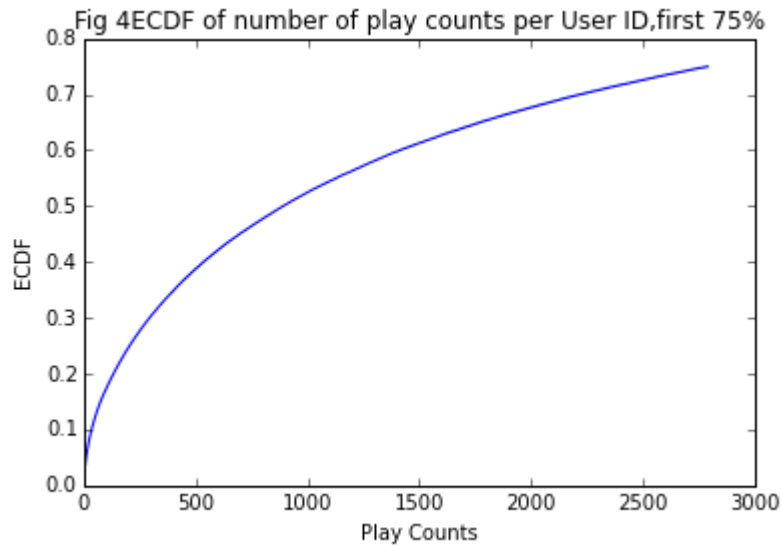
#Focus on first 50% users
plt.plot(Y[0:74000],yvals[0:74000])
plt.xlabel('Play Counts')
plt.ylabel('ECDF')
plt.title('Fig 3 ECDF of number of play counts per User ID,first 50%')
plt.show()

#Focus on first 75% users
plt.plot(Y[0:111000],yvals[0:111000])
plt.xlabel('Play Counts')
plt.ylabel('ECDF')
plt.title('Fig 4ECDF of number of play counts per User ID,first 75%')
plt.show()

#Focus on Last 15% users
plt.plot(Y[-14810:-1],yvals[-14810:-1])
plt.xlabel('Play Counts')
plt.ylabel('ECDF')
plt.title('Fig 5 ECDF of number of play counts per User ID, last 10%')
plt.show()
```

```
np.arrange(len(Y):)  
[      0      1      2 ..., 148108 148109 148110]
```





```
In [6]: print("Number of users who only play the song once:",len(Y[Y == 1]))

Z = np.unique(Y)

print (Z[0:20])
print (Z[-20:-1])

Number of users who only play the song once: 1848
[ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20]
[ 96058  97449  98477 102037 111546 122316 134032 135266 136293 143092
 151504 165642 167273 168977 175822 183972 285978 393515 548427]
```

[Group 6]

1.The distribution is not uniform

As is shown in the figures and table above, the distribution of the play counts per user ID is not generally uniform.

In the first **25%, 50%, 75%** of the total data, the curve is relatively flat. i.e. the distribution of the data is fairly uniform.

However, when we include the whole dataset(Fig 1), the last part of the curve is rather gentle, which means the final part of the x axis data points(i.e.play counts) performs **a distinct increase**.

2. outlier points

As we can see in the Fig 5 and the output of Z, the last **5** points(play counts) perform a prominent increase: 183972 285978 393515 548427 674412. Which makes Fig1 looks steep at the first 75% part and gentle at the last part. These are extreme active users, so we can consider making personalized recommendation plan for them

Besides, there are **1848** users who only play the music **once**, so it is hard to recommend music for them based on the current data. In this case we can consider to recommend the music most rated by other users to these unactive users.

Question 2.3

How many play counts for each artist? Plot CDF or ECDF of the result.

Similarly to the previous question, you need to comment and interpret your result: what is the figure telling you?

```
In [7]: # Compute artist popularity
        # We are interested in how many playcounts per artist
        # ATTENTION! Grouping by artistID may be problematic, as stated above.

        artistPopularity = userArtistDF.groupBy('ArtistID').sum('playCount').collect()
```

[Group 6]

As the artistID for the same artist may vary, the result will not be precise. We will fix this problem later in this Notebook.

```

In [8]: pdf = pd.DataFrame(data=artistPopularity)
Y=np.sort(pdf[1])
yvals=np.arange(len(Y))/float(len(Y))

#print("pdf[1]:\n",pdf[1])

print("np.arange(len(Y)):\n",np.arange(len(Y)))
plt.plot(Y,yvals)
plt.xlabel('Play Counts')
plt.ylabel('ECDF')
plt.grid(True,which="both",ls="-")
plt.title('ECDF of number of play counts per Artist ID')
plt.show()

##first 25% of the artists

plt.plot(Y[0:400000],yvals[0:400000])
plt.xlabel('Play Counts')
plt.ylabel('ECDF')
plt.grid(True,which="both",ls="-")
plt.title('ECDF of number of play counts per Artist ID,first 25%')
plt.show()

##first 50% of the artists

plt.plot(Y[0:800000],yvals[0:800000])
plt.xlabel('Play Counts')
plt.ylabel('ECDF')
plt.grid(True,which="both",ls="-")
plt.title('ECDF of number of play counts per Artist ID,first 50%')
plt.show()

##first 75% of the artists

plt.plot(Y[0:1200000],yvals[0:1200000])
plt.xlabel('Play Counts')
plt.ylabel('ECDF')
plt.grid(True,which="both",ls="-")
plt.title('ECDF of number of play counts per Artist ID,first 75%')
plt.show()

##first 90% of the artists

plt.plot(Y[0:1470000],yvals[0:1470000])
plt.xlabel('Play Counts')
plt.ylabel('ECDF')
plt.grid(True,which="both",ls="-")
plt.title('ECDF of number of play counts per Artist ID,first 90%')
plt.show()

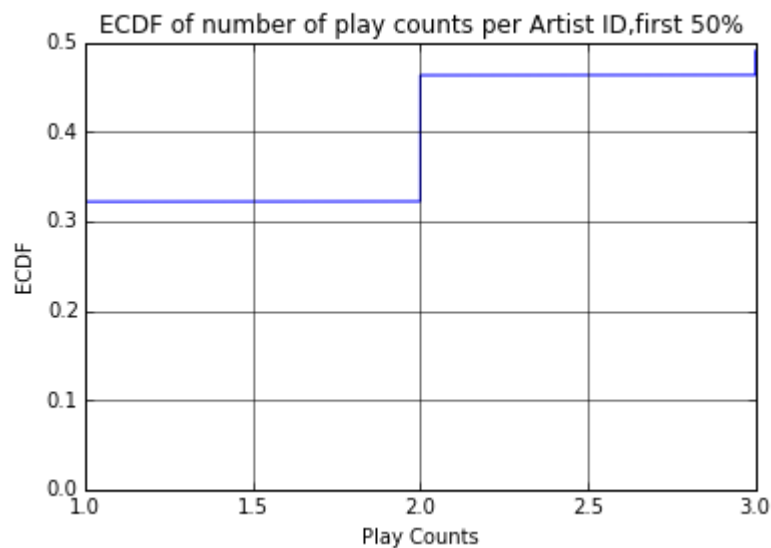
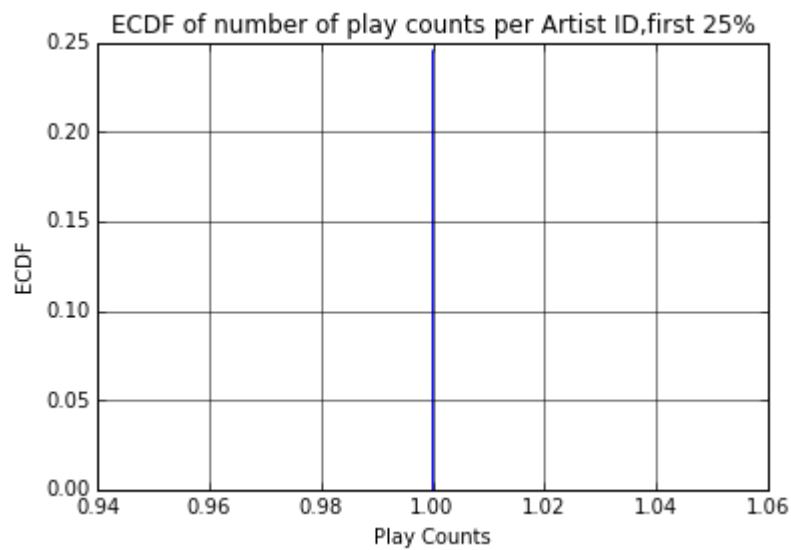
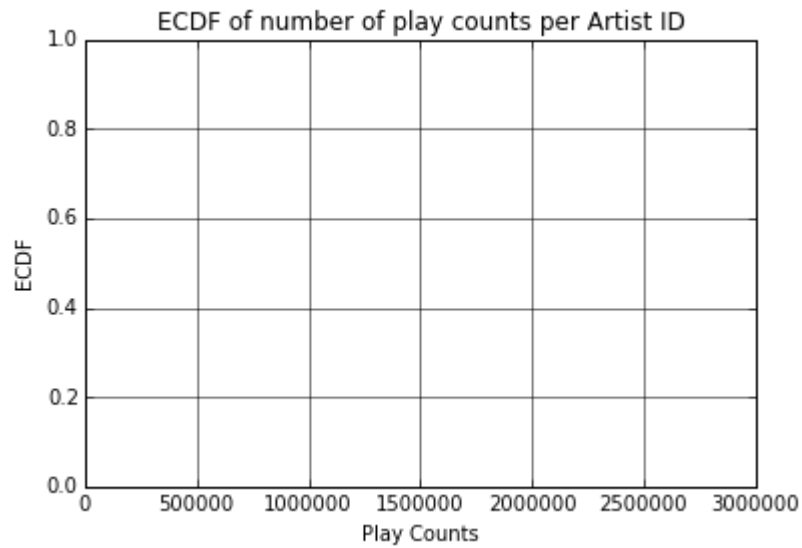
##Last 0.3% of the artists

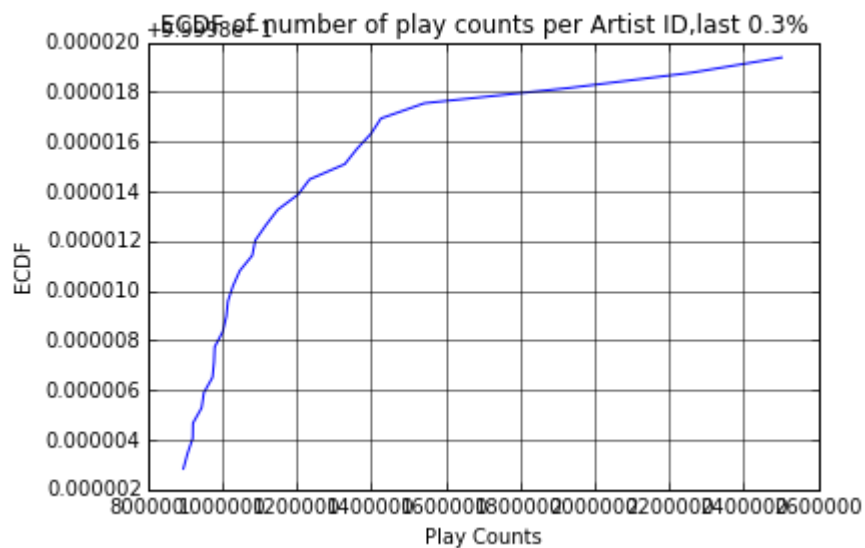
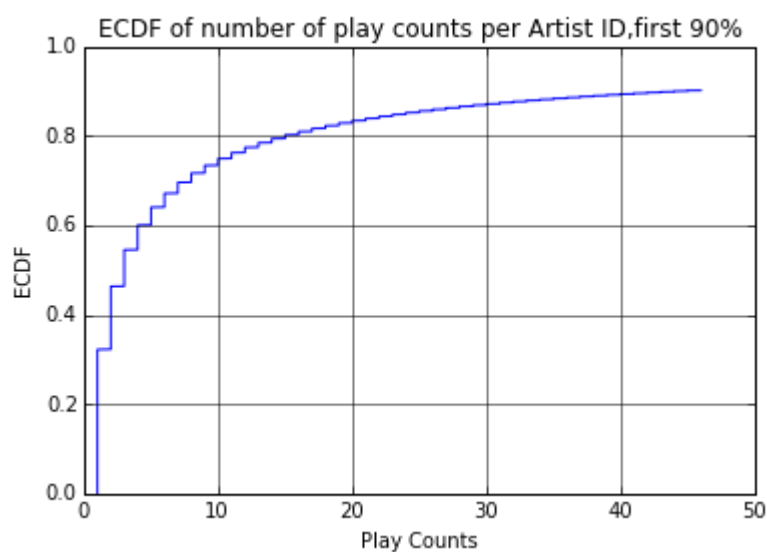
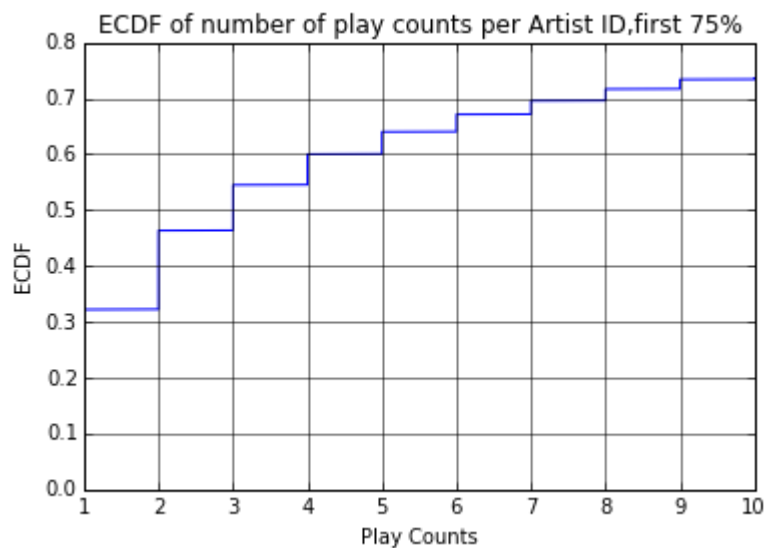
plt.plot(Y[1631000:],yvals[1631000:])
plt.xlabel('Play Counts')
plt.ylabel('ECDF')
plt.grid(True,which="both",ls="-")
plt.title('ECDF of number of play counts per Artist ID,last 0.3%')
plt.show()

```

```
np.arrange(len(Y):)
```

```
[ 0      1      2 ..., 1631025 1631026 1631027]
```






```
In [9]: #print(Y[-1])
W = len(Y[Y==1])
X = float(W)/float(len(Y))
print(W,X)
Z = np.unique(Y,return_counts=True)
Z1 = Z[0]

print (Z[0][0:20],Z[1][0:20])
print (Z[0][-20:],Z[1][-20:])

524942 0.321847325735671
[ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20] [524942 230502
133035 89881 65282 50745 40071 33718 28173 24768
21558 19404 16860 15159 13483 12335 11118 10334 9406 8722]
[ 979539 1001228 1010807 1015064 1028627 1046922 1080412 1088657 1117143
1148568 1203226 1234387 1328869 1361392 1399418 1425942 1542806 1930592
2259185 2502130] [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
```

[Group 6]

Distribution is not uniform

From the figure and data above, the distribution of the artists' popularity is not uniform either, like the distribution of the user activity. There are 524942 artists'songs, nearly 1/3(0.3218) of the whole data, are played only **once**. And there are 524,942+230,502+133,035 = 888,479(**54.47% out of 1,631,028, over half**) artists, whose songs are played **3 times or less**.

We can consider these artists as "**non-popular**" ones, so when we recommend music to users, these artists should be rarely included.

Meanwhile, we can also set a threshold to distinguish which are popular and non-popular artists, say, the artists whose play counts are over **1,600,000**(nearly same as the number of the users) are regarded as **top-popular**(In this case, we got **3 top-poplar artists** among all).From the bar chart below, they are artist **979,1000113 and 4267**.So we recommend these top-popular artists to every users.

These 3 artists are important, because we will use them later in our discussion.

Question 2.4

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

Comment the figure you just obtained:

- are these reasonable results?
- is looking at top-5 artists enough to learn more about your data?
- do you see anything strange in the data?

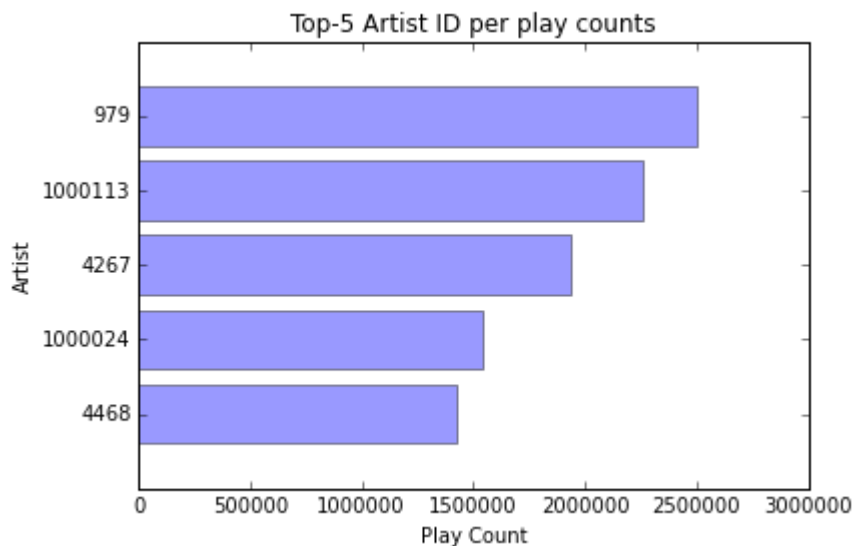
```
In [10]: sortedArtist = sorted(artistPopularity, key = lambda x: -x[1])[:5]
print (sortedArtist[-10:-1])
artistID = [w[0] for w in sortedArtist]

y_pos = range(len(sortedArtist))
frequency = [w[1] for w in sortedArtist]

plt.barh(y_pos, frequency[::-1], align='center', alpha=0.4)
plt.yticks(y_pos, artistID[::-1])
plt.xlabel('Play Count')
plt.ylabel('Artist')
plt.title('Top-5 Artist ID per play counts')
plt.show()

print("ratio of top-5 artists by artists number:",float(5)/1631028)
print("ratio of top-5 artists by play counts:",Z1[-5:].sum()/float(Z1.sum()))

[Row(ArtistID=979, sum(playCount)=2502130), Row(ArtistID=1000113, sum(playCount)=2259185), Row(ArtistID=4267, sum(playCount)=1930592), Row(ArtistID=1000024, sum(playCount)=1542806)]
```



```
ratio of top-5 artists by artists number: 3.0655512964829543e-06
ratio of top-5 artists by play counts: 0.0311906871284
```

[Group 6]

1. These results are resonable.

The play counts of top-5 artists range **from 1,400,000 to 2,500,000**, which are extremely higher than the first 90% artists(**less than 50**) shown in the ECDF figure above.

2. Ratio

These top-5 artists account for only **0.000306%** of the **total artists(1,631,028)**. Meanwhile their play counts occupy **3.12%** of the whole play counts number. This implies these artists are excessively popular than other artists. However, it's not enough only to look at the top 5 artists. There are still **96.88%** play counts are not included, we need to think more about these **"not so popular"** artists in our recommendation system.

All seems clear right now, but ... wait a second! What about the problems indicated above about artist "disambiguation"? Are these artist ID we are using referring to unique artists? How can we make sure that such "opaque" identifiers point to different bands? Let's try to use some additional dataset to answer this question: artist_data.txt dataset. This time, the schema of the dataset consists in:

```
artist ID: long int
name: string
```

We will try to find whether a single artist has two different IDs.

Question 3

Question 3.1

Load the data from /datasets/lastfm/artist_data.txt and use the SparkSQL API to show 5 samples.

HINT: If you encounter some error when parsing lines in data because of invalid entries, parameter mode='DROPMALFORMED' will help you to eliminate these entries. The suggested syntax is:

```
<df>.options(header='false', delimiter='\t', mode='DROPMALFORMED').
```

```
In [8]: customSchemaArtist = StructType([ \
    StructField("artistID", LongType(), True), \
    StructField("name", StringType(), True)])
LongType(), 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)

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

userArtistDF.show()
```

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

only showing top 5 rows

```
+-----+-----+-----+
| 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

Question 3.2

Find 20 artists whose name contains "Aerosmith". Take a look at artists that have ID equal to 1000010 and 2082323. In your opinion, are they pointing to the same artist?

HINT: Function `locate(sub_string, string)` can be useful in this case.

```
In [9]: # get artists whose name contains "Aerosmith"
artistDF[locate("Aerosmith",artistDF.name) > 0].show()

# show two examples
artistDF[artistDF.artistID==1000010].show()
artistDF[artistDF.artistID==2082323].show()
```

```
+-----+-----+
|artistID|          name|
+-----+-----+
|10586006|Dusty Springfield...|
| 6946007|  Aerosmith/RunDMC|
|10475683|Aerosmith: Just P...|
| 1083031|  Aerosmith/ G n R|
| 6872848|Britney, Nsync, N...|
|10586963|Green Day - Oasis...|
|10028830|The Aerosmith Ant...|
|10300357| Run-DMC + Aerosmith|
| 2027746|Aerosmith by Musi...|
| 1140418|[rap]Run DMC and ...|
|10237208| Aerosmith + Run DMC|
|10588537|Aerosmith, Kid Ro...|
| 9934757|Aerosmith - Big Ones|
|10437510|Green Day ft. Oas...|
| 6936680| RUN DNC & Aerosmith|
|10479781|  Aerosmith Hits|
|10114147|Charlies Angels -...|
| 1262439|Kid Rock, Run DMC...|
| 7032554|Aerosmith & Run-D...|
|10033592|  Aerosmith?|
+-----+-----+
```

only showing top 20 rows

```
+-----+-----+
|artistID|    name|
+-----+-----+
| 1000010|Aerosmith|
+-----+-----+
```

```
+-----+-----+
|artistID|          name|
+-----+-----+
| 2082323|01 Aerosmith|
+-----+-----+
```

To answer this question correctly, we need to use an additional dataset `artist_alias.txt` which contains the ids of misspelled artists and standard artists. The schema of the dataset consists in:

```
misspelledID ID: long int
standard ID: long int
```

Question 3.3

Using SparkSQL API, load the dataset from `/datasets/lastfm/artist_alias.txt` then show 5 samples.

```
In [10]: customSchemaArtistAlias = StructType([ \
        StructField("misID",LongType(), True), \
        StructField("stdID",LongType(), True)])

artistAliasDF = sqlContext.read \
    .format('com.databricks.spark.csv') \
    .options(header='false', delimiter='\t',mode='DROPMALFORMED') \
    .load(base + "artist_alias.txt", schema = customSchemaArtistAlias) \
    .cache()

artistAliasDF.show(5)
```

```
+-----+-----+
|  misID|  stdID|
+-----+-----+
| 1092764|1000311|
| 1095122|1000557|
| 6708070|1007267|
|10088054|1042317|
| 1195917|1042317|
+-----+-----+
only showing top 5 rows
```

Question 3.4

Verify the answer of question 3.2 ("Are artists that have ID equal to 1000010 and 2082323 the same ?") by finding the standard ids corresponding to the misspelled ids 1000010 and 2082323 respectively.

```
In [11]: artistAliasDF[artistAliasDF.misID==1000010].show()
artistAliasDF[artistAliasDF.misID==2082323].show()
artistAliasDF[artistAliasDF.misID==10033592].show()

# 1000010 is a standard id, so it haven't been considered as misspelled id in the dataset
```

misID	stdID
2082323	1000010

misID	stdID
2082323	1000010

[Group 6]

From the results above, we found the ID 2082323 and 1000010 point to the same artist. In fact 1000010 is the standard artist ID, and 2082323 is one misspelled ID of 1000010.

Question 4

The misspelled or nonstandard information about artist make our results in the previous queries a bit "sloppy". To overcome this problem, we can replace all misspelled artist ids by the corresponding standard ids and re-compute the basic descriptive statistics on the "amended" data. First, we construct a "dictionary" that maps non-standard ids to a standard ones. Then this "dictionary" will be used to replace the misspelled artists.

Question 4.1

From data in the dataframe loaded from /datasets/lastfm/artist_alias.txt, construct a dictionary that maps each non-standard id to its standard id.

HINT: Instead of using function collect, we can use collectAsMap to convert the collected data to a dictionary inline.

```
In [12]: artistAlias = artistAliasDF.rdd.map(lambda row: ( row[0], row[1])).collectAsMap()
print(type(artistAliasDF))
print(type(artistAliasDF.rdd))
print(type(artistAlias))
artistAliasDF.rdd.map(lambda x:x)

<class 'pyspark.sql.dataframe.DataFrame'>
<class 'pyspark.rdd.RDD'>
<class 'dict'>
```

```
Out[12]: PythonRDD[82] at RDD at PythonRDD.scala:48
```

Question 4.2

Using the constructed dictionary in question 4.1, replace the non-standard artist ids in the dataframe that was loaded from /datasets/lastfm/user_artist_data.txt by the corresponding standard ids then show 5 samples.

NOTE 1: If an id doesn't exist in the dictionary as a misspelled id, it is really a standard id.

Using function map on Spark Dataframe will give us an RDD. We can convert this RDD back to Dataframe by using `sqlContext.createDataFrame(rdd_name, sql_schema)`

NOTE 2: be careful! you need to be able to verify that you indeed solved the problem of having bad artist IDs. In principle, for the new data to be correct, we should have duplicate pairs (user, artist), potentially with different play counts, right? In answering the question, please **show** that you indeed fixed the problem.


```
In [26]: from time import time

def replaceMisspelledIDs(fields):
    finalID = artistAlias.get(fields[1], fields[1])
    return (fields[0], finalID, fields[2])

t0 = time()

newUserArtistDF = sqlContext.createDataFrame(
    userArtistDF.rdd.map( lambda x:replaceMisspelledIDs(x),userArtistDF),
    userArtistDataSchema
)
#artistAliasDF.show()

#newUserArtistDF.rdd.filter(lambda x:x == newUserArtistDF.artistID,)

#newUserArtistDF.[newUserArtistDF.artistID==artistAliasDF.stdID].show()
t1 = time()
print("1st print")
print('The script takes %f seconds' %(t1-t0))
print("2nd print")
print(newUserArtistDF[newUserArtistDF.artistID==1000010].groupBy("artistID").sum("playCount").take(1))
print("3rd print")
print(userArtistDF[userArtistDF.artistID==1000010].groupBy("artistID").sum("playCount").take(1))
print("1st show")
newUserArtistDF[newUserArtistDF.artistID==10300357].show()
print("2nd show")
artistAliasDF[artistAliasDF.misID==10300357].show()
print("3rd show")
artistDF[artistDF.artistID==1008164].show()
```

```

1st print
The script takes 0.940128 seconds
2nd print
[Row(artistID=1000010, sum(playCount)=272992)]
3rd print
[Row(artistID=1000010, sum(playCount)=272606)]
1st show
+-----+-----+-----+
|userID|artistID|playCount|
+-----+-----+-----+
+-----+-----+-----+

2nd show
+-----+-----+
|  misID|  stdID|
+-----+-----+
|10300357|1008164|
+-----+-----+

3rd show
+-----+-----+
|artistID|      name|
+-----+-----+
| 1008164|Run-D.M.C.|
+-----+-----+

```

[Group 6]

In order to prove we really solve this problem, we count the standard artistID **"1000010"** both in userArtistDF and newUserArtistDF. The result shows that the total playCount of these two are 272,992 and 272,606 respectively. So there are $272,992 - 272,606 = 386$ play counts of **artist 1000010** misspelled.


Example

As is shown in **Question3.4**, 10300357 is a misspelled ID. So here we didn't find any artistID equal to 10300357(referring to the 1st show). Instead, in the original Dataframe **"artistDF"**, we can find 10300357 referring to **"Run-DMC + Aerosmith"**, and this has been proved to be a misspelled ID(referring to the 2nd show). Actually, ID 10300357 represents "Run DMC+Aerosmith". Its std ID is 1008164, which represents "Run-MC"

Another way to prove this:

```
In [13]: fullddf=userArtistDF.join(artistAliasDF,userArtistDF['artistID']==artistAliasDF
isID'],'inner')
fullddf.show()

#to figure out how many playCounts is added by misID for each stdID
extraPlayCount=fullddf.groupBy('stdID').sum('playCount')
extraPlayCount.show()
```



userID	artistID	playCount	misID	stdID
1000002	1000434	89	1000434	1000518
1000002	1000762	1	1000762	1001514
1000002	1001220	1	1001220	721
1000002	1001410	5	1001410	1034635
1000002	1002498	1	1002498	3066
1000002	1003377	1	1003377	6691692
1000002	1003633	1	1003633	1237611
1000002	1006102	4	1006102	1034635
1000002	1007652	1	1007652	1001172
1000002	1010219	2	1010219	1008391
1000002	1017405	1	1017405	2006683
1000002	1059598	2	1059598	1000840
1000002	3197	2	3197	2058809
1000002	5702	76	5702	1066440
1000002	709	2	709	2003588
1000019	1000287	1	1000287	1239413
1000019	1000586	1	1000586	2001739
1000019	1000943	6	1000943	1247540
1000019	1001379	4	1001379	1049809
1000019	1002143	1	1002143	4377

only showing top 20 rows

stdID	sum(playCount)
1002519	31
1279485	628
1008798	46589
1279464	252
1233634	200
10407516	1372
1019016	3082
1087201	22
1277866	904
1007205	1492
1007049	129
1019906	584
6915893	59
1287895	749
1014330	48
1172061	23
1314871	54
1061659	188
1004346	14
1009366	1

only showing top 20 rows

Test it with "stdID=1000010":

```
In [16]: print(extraPlayCount[extraPlayCount[0]==1000010].take(1))  
[Row(stdID=1000010, sum(playCount)=386)]
```

Which is same with what we got before, so we indeed fixed the problem.

Question 4.3

Spark actions are executed through a set of stages, separated by distributed "shuffle" operations. Spark can be instructed to **automatically and efficiently** broadcast common data needed by tasks within **each stage**. The data broadcasted this way is cached in **serialized form** and deserialized before running each task.

We can thus improve our answer to question 4.2: we can reduce the communication cost by shipping the "dictionary" in a more efficient way by using broadcast variable. Broadcast variables allow the programmer to keep a read-only variable cached on **each machine** rather than shipping a copy of it with tasks. They are cached in deserialized form. They can be used, for example, to give every node a copy of a large input dataset in an efficient manner.

The broadcast of variable v can be created by $bV = sc.broadcast(v)$. Then value of this broadcast variable can be access via $bV.value$

To question is then: using a broadcast variable, modify the script in question 4.2 to get better performance in terms of running time.

```
In [17]: from time import time

bArtistAlias = sc.broadcast(artistAlias)

def replaceMisspelledIDs(fields):
    finalID = bArtistAlias.value.get(fields[1] , fields[1])
    return (fields[0] , finalID, fields[2])

t0 = time()

newUserArtistDF = sqlContext.createDataFrame(
    userArtistDF.rdd.map(replaceMisspelledIDs),
    userArtistDataSchema
)
newUserArtistDF.show(5)
t1 = time()

print('The script takes %f seconds' %(t1-t0))
newUserArtistDF = newUserArtistDF.cache()
```

```
+-----+-----+-----+
| userID|artistID|playCount|
+-----+-----+-----+
|1000002|      1|      55|
|1000002| 1000006|      33|
|1000002| 1000007|       8|
|1000002| 1000009|     144|
|1000002| 1000010|     314|
+-----+-----+-----+
only showing top 5 rows
```

The script takes 0.482129 seconds

[Group 6]

Compared with 1.529793 seconds we got before, broadcast indeed has better performance.

Although having 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.

Question 5

Well, our data frame contains clean and "standard" data. We can use it to redo previous statistic queries.

Question 5.1

How many unique artists? Compare with the result when using old data.

```
In [18]: uniqueArtists = newUserArtistDF.select("ArtistID").distinct().count()
#uniqueArtists = newUserArtistDF.groupBy("ArtistID").count().count()

print("Total n. of artists: ", uniqueArtists)
```

Total n. of artists: 1568126

[Group 6]

With related to **Question 1.2**, the number of unique artists is 1,631,028, which is more than 1,568,126. This proves that we have $1,631,028 - 1,568,126 = 62,902$ artists' names are misspelled.

Question 5.2

Who are the top-10 artists?

- In terms of absolute play counts
- In terms of "audience size", that is, how many users listened to one of their track at least once

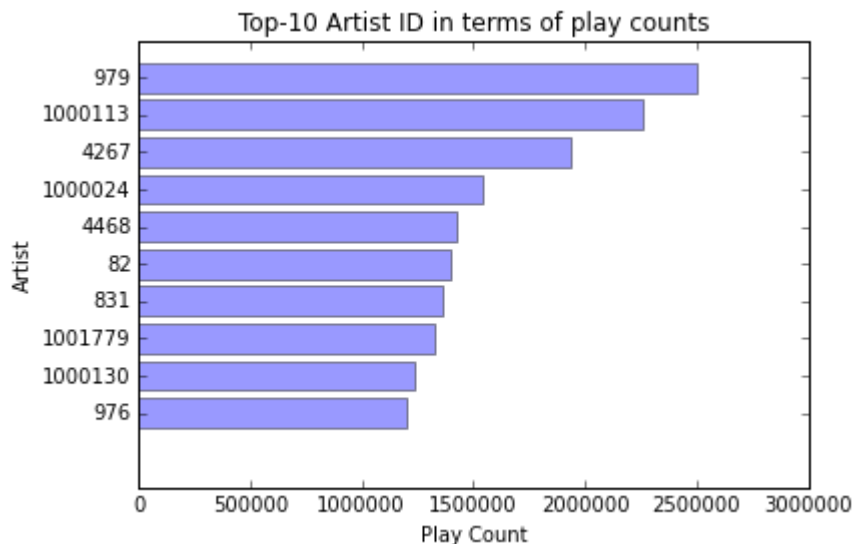
Plot the results, and explain the figures you obtain.

```
In [20]: # calculate top-10 artists in term of play counts
top10ArtistsPC =
newUserArtistDF.groupby("ArtistID").sum("PlayCount").orderBy('sum(playCount)',
    ascending=0).take(10)
y_pos = range(len(top10ArtistsPC))
pdf = pd.DataFrame(data=top10ArtistsPC, columns = ["artistID", "playCount"])

print (pdf)

artistID = [w[0] for w in top10ArtistsPC]
plt.barh(y_pos, pdf.playCount[::-1], align='center', alpha=0.4)
plt.yticks(y_pos, artistID[::-1])
plt.xlabel('Play Count')
plt.ylabel('Artist')
plt.title('Top-10 Artist ID in terms of play counts')
plt.show()
```

	artistID	playCount
0	979	2502596
1	1000113	2259825
2	4267	1931143
3	1000024	1543430
4	4468	1426254
5	82	1399665
6	831	1361977
7	1001779	1328969
8	1000130	1234773
9	976	1203348



[Group 6]

We compared the result with the one before replacing the misspelled artistID, **as shown below**. The play counts after replacement are **slightly larger** than the previous ones.

e.g. The play counts for **artist 979** now is **2,502,596**, while the previous is **2,502,130**.

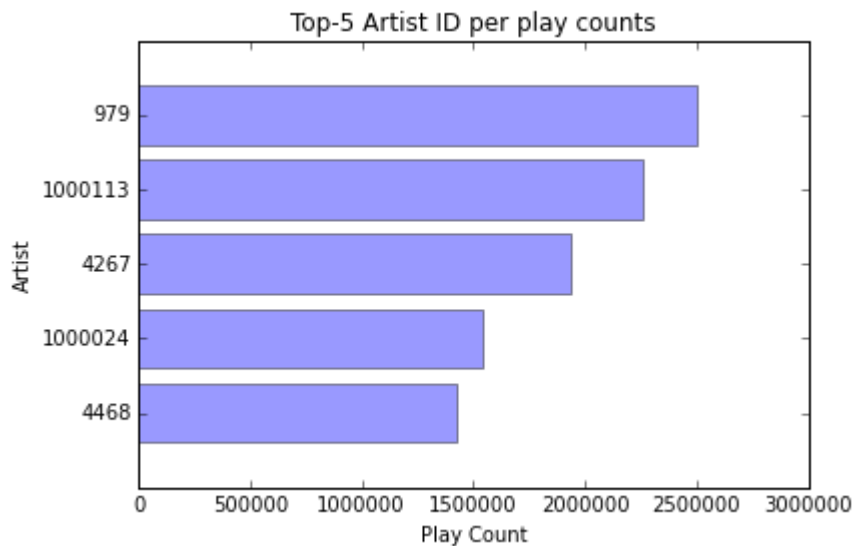
This also proves that some misspelled artist IDs are added into the standard artistIDs after replacement.


```
In [21]: sortedArtist = sorted(artistPopularity, key = lambda x: -x[1])[:5]
print(type(artistPopularity))
print (sortedArtist[-10:-1])
artistID = [w[0] for w in sortedArtist]

y_pos = range(len(sortedArtist))
frequency = [w[1] for w in sortedArtist]

plt.barh(y_pos, frequency[::-1], align='center', alpha=0.4)
plt.yticks(y_pos, artistID[::-1])
plt.xlabel('Play Count')
plt.ylabel('Artist')
plt.title('Top-5 Artist ID per play counts')
plt.show()

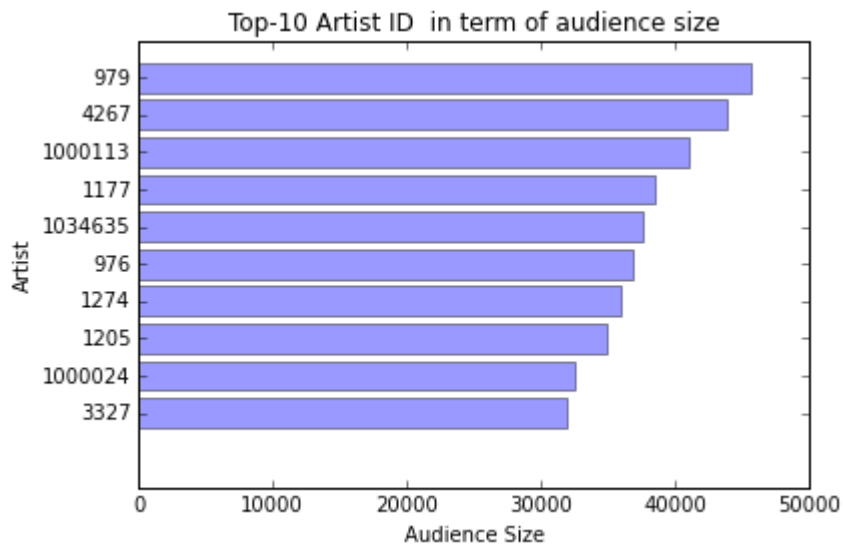
<class 'list'>
[Row(ArtistID=979, sum(playCount)=2502130), Row(ArtistID=1000113, sum(playCount)=2259185), Row(ArtistID=4267, sum(playCount)=1930592), Row(ArtistID=1000024, sum(playCount)=1542806)]
```



```
In [22]: # calculate top-10 artists in term of audience size
top10ArtistsAS = newUserArtistDF.groupby("ArtistID").count().orderBy('count',
ascending=0).take(10)

y_pos = range(len(top10ArtistsAS))
pdf = pd.DataFrame(data=top10ArtistsAS)
artistID = [w[0] for w in top10ArtistsAS]
plt.barh(y_pos, pdf[1][::-1], align='center', alpha=0.4)
plt.yticks(y_pos, artistID[::-1])
print(pdf)
plt.xlabel('Audience Size')
plt.ylabel('Artist')
plt.title('Top-10 Artist ID in term of audience size')
plt.show()
```

	0	1
0	979	45714
1	4267	43943
2	1000113	41063
3	1177	38553
4	1034635	37691
5	976	36899
6	1274	36052
7	1205	34936
8	1000024	32554
9	3327	31983



[Group 6]

```
In [23]: pdfPC = pd.DataFrame(data=top10ArtistsPC,columns=["artistID","playCount"])
pdfAS = pd.DataFrame(data=top10ArtistsAS,columns=["artistID","AudienceSize"])

fulldfArtist=pd.merge(pdfPC,pdfAS,left_on="artistID",right_on="artistID",how='outer')
print(fulldfArtist)
```

	artistID	playCount	AudienceSize
0	979	2502596	45714
1	1000113	2259825	41063
2	4267	1931143	43943
3	1000024	1543430	32554
4	4468	1426254	NaN
5	82	1399665	NaN
6	831	1361977	NaN
7	1001779	1328969	NaN
8	1000130	1234773	NaN
9	976	1203348	36899
10	1177	NaN	38553
11	1034635	NaN	37691
12	1274	NaN	36052
13	1205	NaN	34936
14	3327	NaN	31983

Speaking to popularity of an artist, we should consider not only the **play counts**, but also the **audience size**.

Here we use join function to compare the top-10 artist we got in terms of play counts and audience size respectively.

From the table above we can see that "**1177, 1034635, 1274, 1205, 3327**", these 5 artists have a wider range of fans, but the play counts of the are not in the top 10 list. Meanwhile, "**4468,82,831,1001779,1000130**", these 5 artists stands out in the play counts but have a relatively limited audience range.

Also we can see the rest of the artists in this joint table(namely **979,1000113,4267,1000024,976**), they have both the high play counts and wide audience range, which means they can be regarded as **the top-5 most popular singers** among all the audiences in the sample.

Looking back to our comment in **question 2.3**, the top-popular we recommended in Q2.3 are **979,1000113 and 4267**. Here they are proved to be the real popular artists(high-rated in both play counts and audience size). So it is resonable to recommend them to every users in this system.

Curious about these **three top-popular artists**, we use artistDF to find who they are. Here we go:

```
In [24]: print(artistDF[artistDF.artistID==979].take(1))
print(artistDF[artistDF.artistID==1000113].take(1))
print(artistDF[artistDF.artistID==4267].take(1))
from IPython.display import Image
from IPython.core.display import HTML
from IPython.display import display
print("Radiohead:")
x = Image(url= "http://i0.kym-cdn.com/entries/icons/original/000/018/291/radiohead.jpg")
display(x)
print("The Beatles:")
y = Image(url= "https://im.novinky.cz/051/200511-original1-yn12t.jpg")
display(y)
print("Green Day")
z = Image(url= "http://total-management.com/wp-content/uploads/2017/01/greenday-2.jpg")
display(z)
```

```
[Row(artistID=979, name='Radiohead')]  
[Row(artistID=1000113, name='The Beatles')]  
[Row(artistID=4267, name='Green Day')]  
Radiohead:
```



The Beatles:



Green Day



Question 5.3

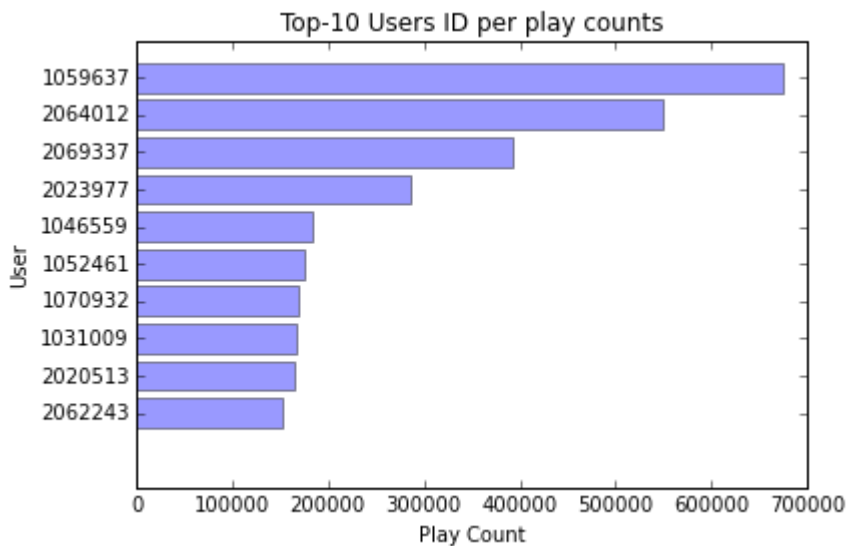
Who are the top-10 users?

- In terms of absolute play counts
- In terms of "curiosity", that is, how many different artists they listened to

Plot the results

```
In [25]: # calculate top 10 users interm of play counts
top10UsersByPlayCount = newUserArtistDF.groupby("UserID").sum("PlayCount").ord
erBy("sum(PlayCount)",ascending=0).take(10)
y_pos = range(len(top10UsersByPlayCount))
pdfUserPC = pd.DataFrame(data=top10UsersByPlayCount,columns = ["userID","playC
ount"])
print(pdfUserPC)
userListPC = [w[0] for w in top10UsersByPlayCount]
plt.barh(y_pos, pdfUserPC.playCount[:, -1], align='center', alpha=0.4)
plt.yticks(y_pos, userListPC[:, -1])
plt.xlabel('Play Count')
plt.ylabel('User')
plt.title('Top-10 Users ID per play counts')
plt.show()
```

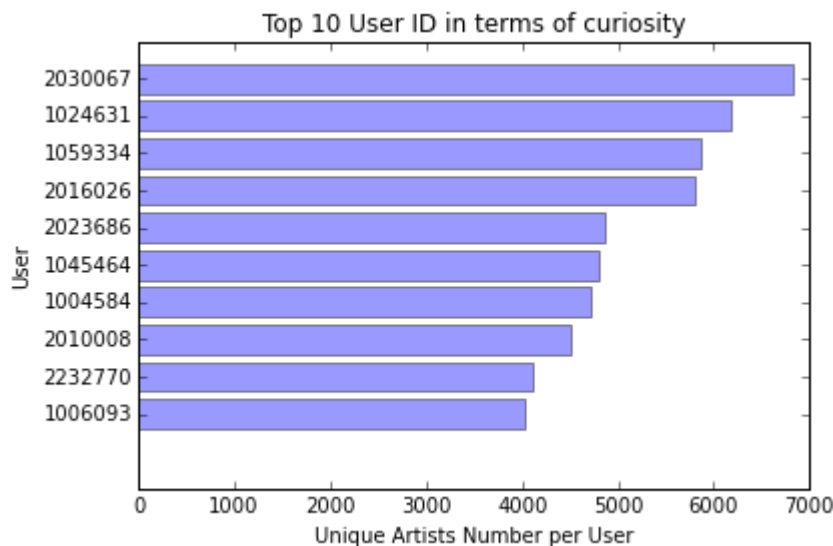
	userID	playCount
0	1059637	674412
1	2064012	548427
2	2069337	393515
3	2023977	285978
4	1046559	183972
5	1052461	175822
6	1070932	168977
7	1031009	167273
8	2020513	165642
9	2062243	151504




```
In [26]: # calculate top 10 users interm of curiosity
top10UsersByCuriosity = newUserArtistDF.groupby("UserID").count().orderBy("count",ascending=0).take(10)
print (top10UsersByCuriosity)

y_pos = range(len(top10UsersByCuriosity))
pdfUserCuri = pd.DataFrame(data=top10UsersByCuriosity,columns = ["userID","artistNumber"])
userListCuri = [w[0] for w in top10UsersByCuriosity]
plt.barh(y_pos,pdfUserCuri.artistNumber[:, -1],align='center',alpha=0.4)
plt.yticks(y_pos,userListCuri[:, -1])
plt.xlabel("Unique Artists Number per User")
plt.ylabel("User")
plt.title("Top 10 User ID in terms of curiosity")
plt.show()
```

```
[Row(UserID=2030067, count=6836), Row(UserID=1024631, count=6188), Row(UserID=1059334, count=5864), Row(UserID=2016026, count=5806), Row(UserID=2023686, count=4863), Row(UserID=1045464, count=4796), Row(UserID=1004584, count=4709), Row(UserID=2010008, count=4512), Row(UserID=2232770, count=4115), Row(UserID=1006093, count=4019)]
```



[Group 6]


```
In [27]: fullddfUser = pd.merge(pdfUserPC,pdfUserCuri,left_on = "userID", right_on="user
ID",how="outer")
print(fullddfUser)
```

	userID	playCount	artistNumber
0	1059637	674412	NaN
1	2064012	548427	NaN
2	2069337	393515	NaN
3	2023977	285978	NaN
4	1046559	183972	NaN
5	1052461	175822	NaN
6	1070932	168977	NaN
7	1031009	167273	NaN
8	2020513	165642	NaN
9	2062243	151504	NaN
10	2030067	NaN	6836
11	1024631	NaN	6188
12	1059334	NaN	5864
13	2016026	NaN	5806
14	2023686	NaN	4863
15	1045464	NaN	4796
16	1004584	NaN	4709
17	2010008	NaN	4512
18	2232770	NaN	4115
19	1006093	NaN	4019

Same as we analyse the artist data, we join the userPlayCount and userCuri by column userID together.

From the table above we saw that the users who are active both in play count and curiosity **don't exist**. Either they listen to a limited range of artists for many times, or they spend time trying various artists without repeating their songs a lot.

Now we have some valuable information about the data. It's the time to study how to build a statistical models.

2. Build a statistical models to make recommendations

2.1 Introduction to recommender systems

In a recommendation-system application there are two classes of entities, which we shall refer to as users and items. Users have preferences for certain items, and these preferences must be inferred from the data. The data itself is represented as a preference matrix A , giving for each user-item pair, a value that represents 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 IT3 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, recommender systems can be categorized into two groups:

- **Content-Based** systems focus on properties of items. Similarity of items is determined by measuring the similarity in their properties.
- **Collaborative-Filtering** systems focus on the relationship between users and items. Similarity of items is determined by the similarity of the ratings of those items by the users who have rated both items.

In the usecase of this notebook, artists take the role of items, and users keep the same role as users. Since we have no information about artists, except their names, we cannot build a content-based recommender system.

Therefore, in the rest of this notebook, we only focus on Collaborative-Filtering algorithms.

2.3 Collaborative-Filtering

In this section, we study a member of a broad class of algorithms called latent-factor models. They try to explain observed interactions between large numbers of users and products through a relatively small number of unobserved, underlying reasons. It is analogous to explaining why millions of people buy a particular few of thousands of possible 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.

First, we formulate the learning problem as a matrix completion problem. Then, we will use a type of matrix factorization model to "fill in" the blanks. We are given implicit ratings that users have given certain items (that is, the number of times they played a particular artist) and our goal is to predict their ratings for the rest of the items. Formally, if there are n users and m items, we are given an $n \times m$ matrix R in which the generic entry (u, i) represents the rating for item i by user u . **Matrix R has many missing entries indicating unobserved ratings, and our task is to estimate these unobserved ratings.**

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

2.3.1 Basic idea and an example of Matrix Factorization

For example, given a preference matrix 5x5 as below, we want to approximate this matrix into the product of two smaller matrixes X and Y .

$$M = \begin{bmatrix} 5 & 2 & 4 & 4 & 3 \\ 3 & 1 & 2 & 4 & 1 \\ 2 & & 3 & 1 & 4 \\ 2 & 5 & 4 & 3 & 5 \\ 4 & 4 & 5 & 4 & \end{bmatrix} \approx M' = \begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \\ x_{31} & x_{32} \\ x_{41} & x_{42} \\ x_{51} & x_{52} \end{bmatrix} \times \begin{bmatrix} y_{11} & y_{12} & y_{13} & y_{14} & y_{15} \\ y_{21} & y_{22} & y_{23} & y_{24} & y_{25} \end{bmatrix}$$

M' is an approximation that is as close to M as possible. To calculate how far from M M' is, we often calculate the sum of squared distances of non-empty elements in M and the corresponding elements in M' . In this way, for M' , besides the approximated elements in M , we also have the non-observed elements. Therefore, to see how much does user i like item j , we simply pick up the value of $M'_{i,j}$.

The challenge is how to calculate X and Y . The bad news is that this can't be solved directly for both the best X and best Y at the same time. Fortunately, if Y is known, we can calculate the best of X , and vice versa. It means from the initial values of X and Y in the beginning, we calculate best X according to Y , and then calculate the best Y according to the new X . This process is repeated until the distance from XY to M is converged. It's simple, right?

Let's take an example. To compute the approximation for the above 5x5 matrix M , first, we init the value of X and Y as below.

$$M' = X \times Y = \begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 1 & 1 \\ 1 & 1 \\ 1 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 & 2 \end{bmatrix}$$

With the initial iteration, we calculate the the Root-Mean-Square Error from XY to M .

Consider the first rows of M and XY . We subtract the first row from XY from the entries in the first row of M , to get 3, 0, 2, 2, 1. We square and sum these to get 18.

In the second row, we do the same to get 1, -1, 0, 2, -1, square and sum to get 7.

In the third row, the second column is blank, so that entry is ignored when computing the RMSE. The differences are 0, 1, -1, 2 and the sum of squares is 6.

For the fourth row, the differences are 0, 3, 2, 1, 3 and the sum of squares is 23.

The fifth row has a blank entry in the last column, so the differences are 2, 2, 3, 2 and the sum of squares is 21.

When we sum the sums from each of the five rows, we get $18 + 7 + 6 + 23 + 21 = 75$. So, $RMSE = \sqrt{75/23} = 1.806$ where 23 is the number of non-empty values in M .

Next, with the given value of Y , we calculate X by finding the best value for X_{11} .

$$M' = X \times Y = \begin{bmatrix} x & 1 \\ 1 & 1 \\ 1 & 1 \\ 1 & 1 \\ 1 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} x+1 & x+1 & x+1 & x+1 & x+1 \\ 2 & 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 & 2 \end{bmatrix}$$

Now, to minimize the $RMSE$ we minimize the difference of the first rows

$(5 - (x + 1))^2 + (2 - (x + 1))^2 + (4 - (x + 1))^2 + (4 - (x + 1))^2 + (3 - (x + 1))^2$. By taking the derivative and set that equal to 0, we pick $x = 2.6$

Given the new value of X , we can calculate the best value for Y .

$$M' = X \times Y = \begin{bmatrix} 2.6 & 1 \\ 1 & 1 \\ 1 & 1 \\ 1 & 1 \\ 1 & 1 \end{bmatrix} \times \begin{bmatrix} y & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 3.6 & 3.6 & 3.6 & 3.6 & 3.6 \\ 2 & 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 & 2 \end{bmatrix}$$

By doing the same process as before, we can pick value for $y = 1.617$. After that, we can check if the $RMSE$ is not converged, we continue to update X by Y and vice versa. In this example, for simple, we only update one element of each matrix in each iteration. In practice, we can update a full row or full matrix at once.

2.3.2 Matrix Factorization: Objective and ALS Algorithm on a Single Machine

More formally, in general, we select k latent features, and describe each user u with a k —dimensional vector x_u , and each item i with a k —dimensional vector y_i .

Then, to predict user u 's rating for item i , we do as follows: $r_{ui} \approx x_u^T y_i$.

This can be put, more elegantly, in a matrix form. Let $x_1, \dots, x_n \in \mathbb{R}^k$ be the factors for the users, and $y_1, \dots, y_m \in \mathbb{R}^k$ the factors for the items. The $k \times n$ user matrix X and the $k \times m$ item matrix Y are then defined by:

$$X = \begin{bmatrix} | & & | \\ x_1 & \cdots & x_n \\ | & & | \end{bmatrix}$$

$$Y = \begin{bmatrix} | & & | \\ y_1 & \cdots & y_i \\ | & & | \end{bmatrix}$$

Our goal is to estimate the complete ratings matrix $R \approx X^T Y$. We can formulate this problem as an optimization problem in which we aim to minimize an objective function and find optimal X and Y . In particular, we aim to minimize the least squares error of the observed ratings (and regularize):

$$\min_{X,Y} \sum_{r_{ui} \text{ observed}} (r_{ui} - x_u^T y_i)^2 + \lambda \left(\sum_u \|x_u\|^2 + \sum_i \|y_i\|^2 \right)$$

Notice that this objective is non-convex (because of the $x_u^T y_i$ term); in fact it's NP-hard to optimize. Gradient descent can be used as an approximate approach here, however it turns out to be slow and costs lots of iterations. Note however, that if we fix the set of variables X and treat them as constants, then the objective is a convex function of Y and vice versa. Our approach will therefore be to fix Y and optimize X , then fix X and optimize Y , and repeat until convergence. This approach is known as **ALS (Alternating Least Squares)**. For our objective function, the alternating least squares algorithm can be expressed with this simple pseudo-code:

Initialize X, Y

while(convergence is not true) **do**

for $u = 1 \dots n$ **do**

$$x_u = \left(\sum_{r_{ui} \in r_{u*}} y_i y_i^T + \lambda I_k \right)^{-1} \sum_{r_{ui} \in r_{u*}} r_{ui} y_i$$

end for

for $u = 1 \dots n$ **do**

$$y_i = \left(\sum_{r_{ui} \in r_{*i}} x_u x_u^T + \lambda I_k \right)^{-1} \sum_{r_{ui} \in r_{*i}} r_{ui} x_u$$

end for

end while

For a single machine, we can analyze the computational cost of this algorithm. Updating each x_u will cost $O(n_u k^2 + k^3)$, where n_u is the number of items rated by user u , and similarly updating each y_i will cost $O(n_i k^2 + k^3)$, where n_i is the number of users that have rated item i .

Once we've computed the matrices X and Y , there are several ways compute a prediction. The first is to do what was discussed before, which is to simply predict $r_{ui} \approx x_u^T y_i$ for each user u and item i . This approach will cost $O(nmk)$ if we'd like to estimate every user-item pair.

However, this approach is prohibitively expensive for most real-world datasets. A second (and more holistic) approach is to use the x_u and y_i as features in another learning algorithm, incorporating these features with others that are relevant to the prediction task.

2.3.3 Parallel Alternating Least Squares

There are several ways to distribute the computation of the ALS algorithm depending on how data is partitioned.

Method 1: using joins

First we consider a fully distributed version, in the sense that all data (both input and output) is stored in a distributed file system. In practice, input data (ratings) and parameters (X and Y) are stored in an a Spark RDD. Specifically, ratings -- that are always **sparse** -- are stored as RDD of triplets:

Ratings: $\text{RDD}((u, i, r_{ui}), \dots)$

Instead, we can use dense representation for factor matrices X and Y , and these are stored as RDDs of vectors. More precisely, we can use the data types introduced in Spark MLlib to store such vectors and matrices:

$X : \text{RDD}(x_1, \dots, x_n)$

$Y : \text{RDD}(y_1, \dots, y_m)$

Now, recall the expression to compute x_u :

$$x_u = \left(\sum_{r_{ui} \in r_{u*}} y_i y_i^T + \lambda I_k \right)^{-1} \sum_{r_{ui} \in r_{u*}} r_{ui} y_i$$

Let's call the first summation *part A* and the second summation *part B*. To compute such parts, in parallel, we can proceed with the following high-level pseudocode:

- Join the Ratings RDD with the Y matrix RDD using key i (items)
- Map to compute $y_i y_i^T$ and emit using key u (user)
- ReduceByKey u (user) to compute $\square \sum_{r_{ui} \in r_{u*}} y_i y_i^T$
- Invert
- Another ReduceByKey u (user) to compute $\square \sum_{r_{ui} \in r_{u*}} r_{ui} y_i$

We can use the same template to compute y_i .

This approach works fine, but note it requires computing $y_i y_i^T$ for each user that has rated item i .

Method 2: using broadcast variables (advanced topic)

The next approach takes advantage of the fact that the 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 R_1 , and similarly partition the Ratings RDD **by item** to create R_2 . This means there are two copies of the same Ratings RDD, albeit with different partitionings. In R_1 , all ratings by the same user are on the same machine, and in R_2 all ratings for same item are on the same machine.
- Broadcast the matrices X and Y . Note that these matrices are not RDD of vectors: they are now "local" matrices.
- Using R_1 and Y , we can use expression x_u from above to compute the update of x_u locally on each machine

- Using R_2 and X , 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 X and Y 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**. It 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 that machine 1 is responsible for users 1,2,...,37: these will be 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> (<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> (<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> (<http://stanford.edu/~rezab/papers/fastals.pdf>)

3. Usecase : Music recommender system

In this usecase, we use the data of users and artists in the previous sections 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. It means that the algorithm should learn without access to user or artist attributes such as age, genre,.... Therefore, an algorithm of type collaborative filtering is the best choice.

Second, in the data, there are some users that have listened to only 1 artist. We need an algorithm that might provide decent recommendations to even these users. After all, at some point, every user starts out with just one play at some point!

Third, we need an algorithm that scales, both in its ability to build large models, and to create recommendations quickly. So, an algorithm which can run on a distributed system (SPARK, Hadoop...) is very suitable.

From these requirement, we can choose using ALS algorithm in SPARK's MLLIB.

Spark MLlib's ALS implementation draws on ideas from 1 (<http://yifanhu.net/PUB/cf.pdf>) and 2 (http://link.springer.com/chapter/10.1007%2F978-3-540-68880-8_32).

3.2 Notes

Currently, MLLIB can only build models from an RDD. That means we have two ways to prepare data:

- Loading to 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 way might not work in all cases. Fortunately, we can use it with this usecase.
- Loading data directly to RDD. However, we have to deal with the invalid data ourselves. In the trade-off, this way is the most reliable, and can work in every case.

In this notebook, we will use the second approach: it requires a bit more effort, but the reward is worth it!

3.3 Cleanup the data

In section 1, we already replaced the ids of misspelled artists 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.

Just as a recall, we work with three datasets in `user_artist_data.txt`, ``` and `artist_alias.txt``. The entries in these file can be empty or have only one field.

In details our goal now 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, but with the artist identifier replaced by its most common alias, as found in the `artist_alias.txt` dataset.

- Since the `artist_alias.txt` file is small, we can use a technique called **broadcast variables** to make such transformation more efficient.

Question 6

Question 6.1

Load data from `/datasets/lastfm/artist_alias.txt` and filter out the invalid entries to construct a dictionary to map from misspelled artists' ids to standard ids.

NOTE: From now on, we will use the "standard" data to train our model.

HINT: If a line contains less than 2 fields or contains invalid numerical values, we can return a special tuple. After that, we can filter out these special tuples.

```
In [23]: 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),rawArtistAlias)

    # fileter out the special tuples
    .filter(lambda x: x!=(-1,-1) )

    # collect result to the driver as a "dictionary"
    .collectAsMap()
)
```

Question 6.2

Using the dictionary in question 6.1, prepare RDD `userArtistDataRDD` by replacing misspelled artists' ids to standard ids. Show 5 samples.

HINT: Using broadcast variable can help us increase the efficiency.

```
In [24]: bArtistAlias = sc.broadcast(artistAlias)
print (type (bArtistAlias))
print (type (bArtistAlias.value.items()))
#print (bArtistAlias.value.items())
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)
print ("type of rawdata",type(rawUserArtistData))
userArtistDataRDD = rawUserArtistData.map(lambda x:disambiguate(x),rawUserArtistData)
userArtistDataRDD.take(5)

<class 'pyspark.broadcast.Broadcast'>
<class 'dict_items'>
type of rawdata <class 'pyspark.rdd.RDD'>

Out[24]: [('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, we must use a preference matrix as an input. MLLIB uses the class `Rating` to support the construction of a distributed preference matrix.

Question 7

Question 7.1

Given RDD `userArtistDataRDD` in question 6.2, construct a new RDD `trainingData` by transforming each item of it into a `Rating` object.

```
In [19]: from pyspark.mllib.recommendation import ALS, MatrixFactorizationModel, Rating
```

```
In [21]: allData = userArtistDataRDD.map(lambda r: Rating(r[0], r[1], r[2])).repartition(10).cache()
print(allData.take(20))
```

```
[Rating(user=2036216, product=1201049, rating=9.0), Rating(user=2036216, product=1203824, rating=4.0), Rating(user=2036216, product=1216872, rating=2.0),
Rating(user=2036216, product=1231740, rating=9.0), Rating(user=2036216, product=1233193, rating=7.0), Rating(user=2036216, product=1233977, rating=42.0),
Rating(user=2036216, product=1234338, rating=5.0), Rating(user=2036216, product=1234422, rating=11.0), Rating(user=2036216, product=1235281, rating=4.0),
Rating(user=2036216, product=1235736, rating=6.0), Rating(user=2036216, product=2153582, rating=11.0), Rating(user=2036216, product=2154365, rating=6.0),
Rating(user=2036216, product=2160296, rating=4.0), Rating(user=2036216, product=2160373, rating=9.0), Rating(user=2036216, product=2194, rating=8.0),
Rating(user=2036216, product=221, rating=27.0), Rating(user=2036216, product=2229, rating=9.0), Rating(user=2036216, product=2237, rating=50.0), Rating(user=2036216, product=227, rating=9.0), Rating(user=2036216, product=2439, rating=15.0)]
```

[Group 6]

```
In [27]: newUserArtistDF[newUserArtistDF.userID==2036216][newUserArtistDF.artistID==1201049].take(1)
```

```
Out[27]: [Row(userID=2036216, artistID=1201049, playCount=9)]
```

The cell above is to check the usage of Rating function.

Question 7.2

A model can be trained by using `ALS.trainImplicit(<training data>, <rank>)`, where:

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

We can also use some additional parameters to adjust the quality of the model. Currently, let's set

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

to build model.

```
In [29]: t0 = time()
model = ALS.trainImplicit( allData,10 )
t1 = time()
print("finish training model in %f secs" % (t1 - t0))
```

```
finish training model in 27.488327 secs
```

```
In [27]: print (type(model))

<class 'pyspark.mllib.recommendation.MatrixFactorizationModel'>
```

Question 7.3

The trained model can be saved into HDFS for later use. This can be done via `model.save(sc, <file_name>)`. Let's use this function to store our model as name `lastfm_model.spark`.

NOTE 1: since you may have noticed that building the model takes some time, it might come to your mind that this information could be stored, such that you can "interrupt" your laboratory session here, and restart next time by loading your model.

NOTE 2: funnily enough, it could take more time to save the model than to build it from scratch! So take a look at the execution time to save the model: this method actually stores the model as Parquet files, which are column-oriented and compressed.

NOTE 3: to check you have your file on HDFS, you are invited to open a terminal from the "Home" Jupyter dashboard, and type `hdfs dfs -ls` to check.

```
In [30]: ! hdfs dfs -rm -R -f -skipTrash lastfm_model.spark
t0 = time()
model.save(sc , "lastfm_model.spark")
t1 = time()
print("finish saving model in %f secs" % (t1-t0))

Deleted lastfm_model.spark
finish saving model in 4.371998 secs
```

[Group 6]

Based on our observation, the **saving time(4.371998s)** is shorter than the **running time(27.488327s)**

```
In [36]: ! hdfs dfs -ls

Found 1 items
drwxr-xr-x  - user6 supergroup          0 2017-03-16 15:50 lastfm_model.spar
k
```

Question 7.4

A saved model can be load from file by using `MatrixFactorizationModel.load(sc, <file_name>)`.

Let's load our model from file.

```
In [31]: t0 = time()
model_loaded = MatrixFactorizationModel.load(sc, "lastfm_model.spark")
t1 = time()
print("finish loading model in %f secs" % (t1 - t0))

finish loading model in 1.337787 secs
```

Question 7.5

Print the first row of user features in our model.

```
In [32]: print(type(model_loaded))
#userFeatures():Returns a paired RDD, where the first element is the user and
the
#second is an array of features corresponding to that user.
print(model.userFeatures().take(1))
print(model.productFeatures().take(1))

<class 'pyspark.mllib.recommendation.MatrixFactorizationModel'>
[(120, array('d', [0.038196176290512085, -0.004820053465664387, -0.0486153811
2163544, -0.021210389211773872, 0.0042330375872552395, -0.034289903938770294,
0.04037928581237793, -0.008364418521523476, 0.014011275954544544, -0.03047269
955277443])))]
[(24, array('d', [0.044969890266656876, 7.692998769925907e-05, 0.026003751903
772354, -0.049107037484645844, 0.029640045017004013, 0.006427656393498182, -
0.012239793315529823, -0.00363758928142488, -0.02331002801656723, 0.022030219
435691833])))]
```

Question 8

Show the top-5 artist names recommended for user 2093760.

HINT: The recommendations can be given by function `recommendProducts(userID, num_recommendations)`. These recommendations are only artist ids. You have to map them to artist names by using data in `artist_data.txt`.

```
In [104]: # Make five recommendations to user 2093760
recommendations = (model.recommendProducts(2093760,5))

# construct set of recommended artists
recArtist = set(recommendations)
print(recommendations)

[Rating(user=2093760, product=1034635, rating=0.3497436397415201), Rating(use
r=2093760, product=2814, rating=0.3457145777297242), Rating(user=2093760, pro
duct=1001819, rating=0.34286126130029515), Rating(user=2093760, product=930,
rating=0.3412789670903478), Rating(user=2093760, product=393, rating=0.33985
18507018743)]
```

Group 6

```
In [105]: artistPlaycount = newUserArtistDF.groupBy("artistID").count().orderBy('count',
        ascending=0)
        userPlayCount = newUserArtistDF.groupBy("userID").sum("PlayCount").orderBy('sum(PlayCount)', ascending=0)

        print(artistPlaycount[artistPlaycount.artistID==1007614].take(1))
        print(artistPlaycount[artistPlaycount.artistID==4605].take(1))
        print(artistPlaycount[artistPlaycount.artistID==2814].take(1))
        print(artistPlaycount[artistPlaycount.artistID==250].take(1))
        print(artistPlaycount[artistPlaycount.artistID==829].take(1))

        [Row(artistID=1007614, count=16825)]
        [Row(artistID=4605, count=16331)]
        [Row(artistID=2814, count=17540)]
        [Row(artistID=250, count=24963)]
        [Row(artistID=829, count=10145)]
```

As is shown in the result above, user **"2093760"** has listened only 5 artists, and the play count in total is 9, which means that 2093760 is an **unactive user** among all the data set.

While applying this recommendation system, the recommendation artists for 2093760 are **1007614,4605,2814,250,829**. We calculate the play count of these 5 recommendation artists : 16825, 16331, 17540, 24963, 10145. They are quite popular artists.

So the result shows that, while recommending artists for unactive users, this system tends to give the relatively popular artists to these users. This consequence also conforms to our common sense.

```
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 [112]: # Filter in those artists, get just artist, and print
def artistNames(line):
    # [artistID, name]
    if (line in recArtist):
        return True
    else:
        return False
#recListID = artistByID.filter(lambda x: artistNames(x)).keys().collect()
recList = artistByID.filter(lambda x: artistNames(x)).values().collect()
#print(recListID[:5])
print(recList)

[]
```

```
In [106]: print(artistByID.take(1))

[(1134999, '06Crazy Life')]
```

```
In [108]: print(recArtist)

{Rating(user=2093760, product=1034635, rating=0.3497436397415201), Rating(user=2093760, product=1001819, rating=0.34286126130029515), Rating(user=2093760, product=393, rating=0.3398518507018743), Rating(user=2093760, product=2814, rating=0.3457145777297242), Rating(user=2093760, product=930, rating=0.3412789670903478)}
```

IMPORTANT NOTE

At the moment, it is necessary to manually unpersist the RDDs inside the model when you are done with it. The following function can be used to make sure models are promptly uncached.

```
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 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 0.9:0.1 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.

Question 9

Question 9.1

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([0.9,0.1],)
         trainData.cache()
         cvData.cache()
```

```
Out[38]: PythonRDD[366] at RDD at PythonRDD.scala:48
```

```
In [39]: t0 = time()
model = ALS.trainImplicit(trainData,rank=10,iterations=5,lambda_=0.01,alpha=1.0)
t1 = time()
print("finish training model in %f secs" % (t1 - t0))

finish training model in 46.793379 secs
```

Area under the ROC curve: a function to compute it

```
In [42]: # Get all unique artistId, and broadcast them
allItemIDs = np.array(allData.map(lambda x: x[1]).distinct().collect())
bAllItemIDs = sc.broadcast(allItemIDs)
```

```

In [40]: 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 t
he user.
# 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. Duplicate
s are OK.
        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)

def ratioOfCorrectRanks(positiveRatings, negativeRatings):

    # find number elements in arr that has index >= start and has value smalle
r than x
    # 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

## 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-
negative
## pairs that are correctly ranked is computed. The result is equal to the
AUC metric.
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

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 gather
by user
    positivePredictions = predictFunction(positiveUserProducts).groupBy(lambda
r: r.user)

    # Create a set of "negative" products for each user. These are randomly ch
osen
    # from among all of the other items, excluding those that are "positive" f
or the user.
    negativeUserProducts = positiveUserProducts.groupByKey().mapPartitions(xtr
actNegative).flatMap(lambda x: x)
    # Make predictions on the rest
    negativePredictions = predictFunction(negativeUserProducts).groupBy(lambda
r: r.user)

```

```

    return (
        positivePredictions.join(negativePredictions)
        .values()
        .map(
            lambda positive_negativeRatings: ratioOfCorrectRanks(positive_negativeRatings[0], positive_negativeRatings[1])
        )
        .mean()
    )

```

Question 9.2

Using part cvData and function calculateAUC to compute the AUC of the trained model.

```

In [43]: t0 = time()
auc = calculateAUC(cvData,bAllItemIDs, model.predictAll)
t1 = time()
print("auc=",auc)
print("finish in %f seconds" % (t1 - t0))

auc= 0.9643956772548857
finish in 35.997382 seconds

```

```

In [44]: print(type(trainData))

<class 'pyspark.rdd.PipelinedRDD'>

```

Question 9.3

Now we have the UAC of our model, it's helpful to benchmark this against a simpler approach. For example, consider recommending the globally most-played artists to every user. This is not personalized, but is simple and may be effective.

Implement this simple popularity-based prediction algorithm, evaluate its AUC score, and compare to the results achieved by the more sophisticated ALS algorithm.

```
In [45]: bListenCount = sc.broadcast(trainData.map(lambda r: (r[1],
r[2])).reduceByKey(lambda x,y:x+y).collectAsMap())
print (trainData.map(lambda r: (r[1],
r[2])).toDF(["artist","rating"]).groupBy("artist").sum("rating").orderBy("arti
st").take(10))
### danger !!! print(trainData.map(lambda r: (r[1], r[2])).reduceByKey(lambda
a x,y:x+y).collectAsMap())
```

```
def predictMostListened(allData):
    #[userID,artistID,rating]=allData.split(" ")
    return allData.map(lambda r: Rating(r[0], r[1],
bListenCount.value.get(r[1], 0.0)))
```

```
[Row(artist=1, sum(rating)=302896.0), Row(artist=2, sum(rating)=628408.0), Ro
w(artist=3, sum(rating)=3896.0), Row(artist=4, sum(rating)=14212.0), Row(arti
st=5, sum(rating)=12490.0), Row(artist=6, sum(rating)=918.0), Row(artist=7, s
um(rating)=177.0), Row(artist=8, sum(rating)=11297.0), Row(artist=9, sum(rati
ng)=1357.0), Row(artist=11, sum(rating)=7569.0)]
```

```
In [46]: auc = calculateAUC(cvData,bAllItemIDs, predictMostListened)
print(auc)
```

```
0.9373245953417353
```

3.6 Personalized recommendations with ALS

In the previous section, we build our models with some given parameters 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 k 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 running 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.

Question 10

Question 10.1

For simplicity, assume that we want to explore the following parameter space: $rank \in \{10, 50\}$, $lambda_ \in \{1.0, 0.0001\}$ and $alpha \in \{1.0, 40.0\}$.

Find the best combination of them in terms of the highest AUC value.

```
In [48]: evaluations = []

for rank in [8,10,50,70]:
    for lambda_ in [5.0,2.0,1.0, 0.0001]:
        for alpha in [1.0, 40.0, 80.0]:
            print("Train model with rank=%d lambda_=%f alpha=%f" % (rank, lambda_, alpha))
            # with each combination of params, we should run multiple times and get avg
            # for simple, we only run one time.
            model = ALS.trainImplicit(trainData,rank=rank,lambda_=lambda_,alpha=alpha)

            auc = calculateAUC(cvData,bAllItemIDs,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=8 lambda_=5.000000 alpha=1.000000
Train model with rank=8 lambda_=5.000000 alpha=40.000000
Train model with rank=8 lambda_=5.000000 alpha=80.000000
Train model with rank=8 lambda_=2.000000 alpha=1.000000
Train model with rank=8 lambda_=2.000000 alpha=40.000000
Train model with rank=8 lambda_=2.000000 alpha=80.000000
Train model with rank=8 lambda_=1.000000 alpha=1.000000
Train model with rank=8 lambda_=1.000000 alpha=40.000000
Train model with rank=8 lambda_=1.000000 alpha=80.000000
Train model with rank=8 lambda_=0.000100 alpha=1.000000
Train model with rank=8 lambda_=0.000100 alpha=40.000000
Train model with rank=8 lambda_=0.000100 alpha=80.000000
Train model with rank=10 lambda_=5.000000 alpha=1.000000
Train model with rank=10 lambda_=5.000000 alpha=40.000000
Train model with rank=10 lambda_=5.000000 alpha=80.000000
Train model with rank=10 lambda_=2.000000 alpha=1.000000
Train model with rank=10 lambda_=2.000000 alpha=40.000000
Train model with rank=10 lambda_=2.000000 alpha=80.000000
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_=1.000000 alpha=80.000000
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=10 lambda_=0.000100 alpha=80.000000
Train model with rank=50 lambda_=5.000000 alpha=1.000000
Train model with rank=50 lambda_=5.000000 alpha=40.000000
Train model with rank=50 lambda_=5.000000 alpha=80.000000
Train model with rank=50 lambda_=2.000000 alpha=1.000000
Train model with rank=50 lambda_=2.000000 alpha=40.000000
Train model with rank=50 lambda_=2.000000 alpha=80.000000
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_=1.000000 alpha=80.000000
Train model with rank=50 lambda_=0.000100 alpha=1.000000
Train model with rank=50 lambda_=0.000100 alpha=40.000000
Train model with rank=50 lambda_=0.000100 alpha=80.000000
Train model with rank=70 lambda_=5.000000 alpha=1.000000
Train model with rank=70 lambda_=5.000000 alpha=40.000000
Train model with rank=70 lambda_=5.000000 alpha=80.000000
Train model with rank=70 lambda_=2.000000 alpha=1.000000
Train model with rank=70 lambda_=2.000000 alpha=40.000000
Train model with rank=70 lambda_=2.000000 alpha=80.000000
Train model with rank=70 lambda_=1.000000 alpha=1.000000
Train model with rank=70 lambda_=1.000000 alpha=40.000000
Train model with rank=70 lambda_=1.000000 alpha=80.000000
Train model with rank=70 lambda_=0.000100 alpha=1.000000
Train model with rank=70 lambda_=0.000100 alpha=40.000000
Train model with rank=70 lambda_=0.000100 alpha=80.000000

```

	0	1
0	(70, 0.0001, 1.0)	0.949382
1	(50, 0.0001, 1.0)	0.953258
2	(8, 0.0001, 1.0)	0.963364
3	(10, 0.0001, 1.0)	0.963929
4	(70, 1.0, 1.0)	0.965697
5	(50, 1.0, 1.0)	0.966908
6	(10, 1.0, 1.0)	0.967891
7	(8, 1.0, 1.0)	0.970663

8	(70, 2.0, 1.0)	0.971100
9	(50, 2.0, 1.0)	0.971183
10	(10, 2.0, 1.0)	0.971719
11	(8, 2.0, 1.0)	0.972288
12	(70, 0.0001, 40.0)	0.973777
13	(10, 0.0001, 40.0)	0.974696
14	(50, 0.0001, 40.0)	0.974904
15	(70, 1.0, 40.0)	0.975240
16	(10, 5.0, 1.0)	0.975358
17	(8, 1.0, 40.0)	0.975363
18	(8, 5.0, 1.0)	0.975945
19	(8, 0.0001, 80.0)	0.975948
20	(70, 0.0001, 80.0)	0.975995
21	(50, 1.0, 40.0)	0.976224
22	(70, 2.0, 40.0)	0.976257
23	(10, 2.0, 40.0)	0.976526
24	(10, 0.0001, 80.0)	0.976683
25	(50, 2.0, 40.0)	0.976686
26	(10, 1.0, 40.0)	0.976749
27	(70, 1.0, 80.0)	0.976910
28	(8, 5.0, 40.0)	0.976924
29	(8, 2.0, 80.0)	0.976940
30	(50, 0.0001, 80.0)	0.977359
31	(8, 0.0001, 40.0)	0.977508
32	(8, 1.0, 80.0)	0.977521
33	(8, 2.0, 40.0)	0.977690
34	(10, 5.0, 40.0)	0.977764
35	(70, 2.0, 80.0)	0.977822
36	(8, 5.0, 80.0)	0.977874
37	(50, 1.0, 80.0)	0.977895
38	(50, 5.0, 40.0)	0.977992
39	(70, 5.0, 40.0)	0.978099
40	(10, 2.0, 80.0)	0.978276
41	(50, 5.0, 1.0)	0.978348
42	(70, 5.0, 1.0)	0.978470
43	(10, 1.0, 80.0)	0.978482
44	(10, 5.0, 80.0)	0.978714
45	(50, 2.0, 80.0)	0.978714
46	(70, 5.0, 80.0)	0.979311
47	(50, 5.0, 80.0)	0.979608

Out[48]: PythonRDD[366] at RDD at PythonRDD.scala:48

[Group 6]

The first parameter **rank** stands for the number of latent factors in the model, more features means finer granularity, but it will cause the problem of overfitting.

The second parameter, **lambda_**, represents for overfitting. Higher value resists overfitting. So to some extent it is complementary with **rank**.

The third parameter, **alpha**, refers to the relation between observed and unobserved userproduct.

Based on the original parameters(rank = 10, 50, lambda = 1, 0.0001, alpha = 1, 40), we add some new parameters, and train for the 48 results in total. The best result we got is **AUC = 0.979608**, with parameters rank = 50, lambda = 5.0, alpha = 80.0. Compared the initial result(rank = 10, lambda_ = 0.01, alpha = 1.0, **AUC = 0.9373245953417353**), the increment in AUC is remarkable.

Question 10.2

Using "optimal" hyper-parameters in question 10.1, re-train the model and show top-5 artist names recommended for user 2093760.

```
In [54]: model = ALS.trainImplicit(trainData,rank=50,lambda_=5.0,alpha=80.0)
allData.unpersist()

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

recommendedProductIDs = set(recommendations)
print (recommendations)
recList = artistByID.filter(lambda x: artistNames(x)).values().collect()
print(recList[0:20])

unpersist(model)
```

```
[Rating(user=2093760, product=1034635, rating=0.3497436397415201), Rating(user=2093760, product=2814, rating=0.3457145777297242), Rating(user=2093760, product=1001819, rating=0.34286126130029515), Rating(user=2093760, product=930, rating=0.3412789670903478), Rating(user=2093760, product=393, rating=0.3398518507018743)]
['06Crazy Life', 'Pang Nakarin', 'Terfel, Bartoli- Mozart: Don', 'The Flaming Sidebur', 'Bodenstandig 3000', 'Jota Quest e Ivete Sangalo', 'Toto_XX (1977', 'U.S Bombs -', 'artist formaly know as Mat', 'Kassierer - Musik für beide Ohren', 'Rahzel, RZA', 'Jon Richardson', 'Young Fresh Fellows', 'The Minus 5', 'Ki-ya-Kiss', 'Underminded - The Task Of Modern Educator', 'Kox-Box', 'alexisonfire [wo!]', 'dj salinger', 'The B52's - Channel Z', '44 Hoes']
```