# MLLIB COMPLETELY ABSTRACTS AWAY THE TECHNICAL IMPLEMENTATION DETAILS OF ALS

Recap

# APACHE SPARK

Spark comes with some additional packages that make it **truly general - purpose**

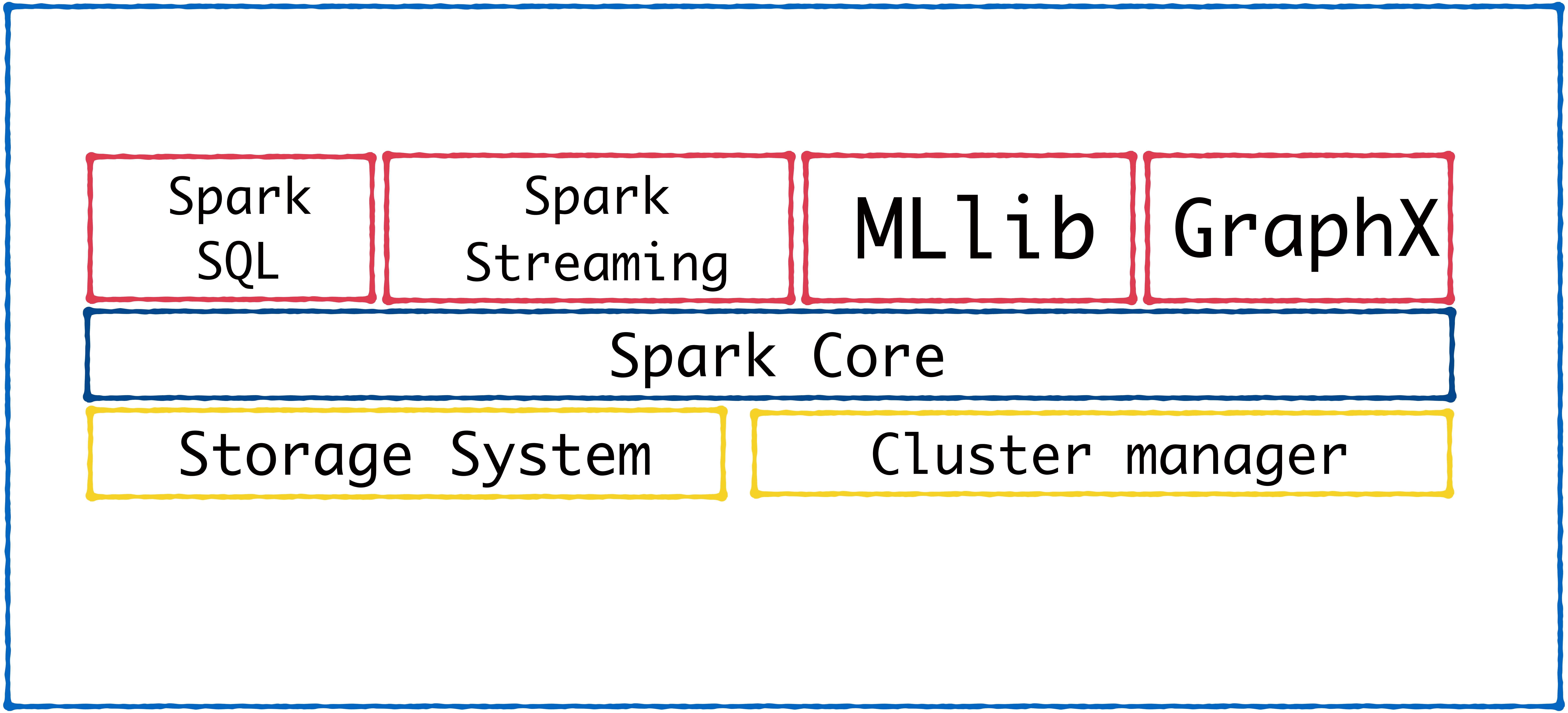
Spark Core

Storage System

Cluster manager

Recap

# APACHE SPARK



Recap

# APACHE SPARK

Spark SQL | Spark Streaming | MLlib | GraphX

Spark Core

Storage System | Cluster Manager

**MLlib provides built-in Machine Learning functionality in Spark**

Recap

# APACHE SPARK

MLlib

GraphX

Spark Core

Cluster manager

This basically solves 2 problems with previous computing frameworks

**Abstraction**

**Performance**

Recap

# APACHE SPARK

MLlib

Abstraction

Machine Learning algorithms are pretty complicated

Recap

# APACHE SPARK

## Abstraction

Python and R have libraries which allow you to plug and play ML algorithms with minimal effort

MLlib

GraphX

Spark Core

These libraries are not suited for distributed computing

Recap

# APACHE SPARK

MLlib

## Abstraction

Hadoop MapReduce is great for distributed computing, but it requires you to **implement the algorithms yourself** as map and reduce tasks

Recap

# APACHE SPARK

**Abstraction**

MLlib

**MLlib has Built-in modules for Classification, regression, clustering, recommendations etc algorithms**

Recap

# APACHE SPARK

MLlib

**Abstraction**

Under the hood the library takes care of running these algorithms **across a cluster**

Recap

# APACHE SPARK

MLlib

## Abstraction

This completely abstracts the programmer from

Implementing the ML algorithm

Intricacies of running it across a cluster

Recap

# APACHE SPARK

Spark SQL | Spark Streaming | MLlib | GraphX

Spark Core

Storage System | Cluster manager

This basically solves 2 problems with previous computing frameworks

**Abstraction**

**Performance**

Recap

# APACHE SPARK

## Performance

Machine Learning algorithms are iterative, which means you need to make multiple passes over the same data

MLlib

Hadoop MapReduce is heavy on disk writes which is not efficient for Machine Learning

Recap

# APACHE SPARK

## Performance

Spark SQL | Spark Streaming | **MLlib** | GraphX

Spark Core

Storage System | Cluster manager

Since Spark's RDDs are **in-memory**

It can make **multiple passes** over the same data **without** doing **disk writes**

ALTERNATING LEAST SQUARES

MLlib

# Let's use ALS to find Artist Recommendations

## using the Audioscrobbler Dataset

# Audioscrobbler Dataset

MLlib

Audioscrobbler is an online music recommendation service

It was acquired by **Last.fm**

# Audioscrobbler Dataset

MLlib

# The dataset has

User ID

Artist ID

# times user listened to the artist

# Audioscrobbler Dataset

MLlib

User ID

Artist ID

# times user listened to the artist

If you consider this as an implicit rating

# Audioscrobbler Dataset

MLlib

User ID | Artist ID | # times user listened to the artist

This dataset can be seen as a USER-PRODUCT-RATING Matrix

# Audioscrobbler Dataset

MLlib

## USER-PRODUCT-RATING MATRIX

Artist ID

User ID

| | PROD 1 | PROD 2 | PROD 3 | PROD 4 | .. .. .. | PROD D |
|---|---|---|---|---|---|---|
| USER 1 | 4 | - | 4 | - | - | - |
| USER 2 | - | 3 | 4 | - | - | - |
| USER 3 | 5 | 3 | 2 | - | - | 5 |
| USER 4 | 2 | - | 2 | - | - | 4 |
| .. | - | - | - | 4 | - | - |
| .. | - | 1 | - | - | - | - |
| USER N | 4 | 3 | 4 | - | - | 5 |

# times user listened to the artist

# Audioscrobbler Dataset

MLlib

## USER-PRODUCT-RATING MATRIX

All you need to do is feed this matrix to ALS in MLlib!

# Let's now look at the code

# Load the dataset with User-Artist ratings

```
val rawUserArtistData = sc.textFile(uadatapath)
```

```
'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'
```

The data in this file looks like this

```
'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'
```

# User ID

```
'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'
```

# Artist ID

```
'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'
```

# times user listened to the artist

Implicit rating

```
'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'
```

Let's get a quick sense of this ratings column

```
rawUserArtistData.map(x => x.split(" ")(2).toDouble).stats()
```

```
'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
```

# Extract the ratings column

```
rawUserArtistData.map(x => x.split(" ")(2).toDouble).stats()
```

```
[u'1000002 1 55',
 u'1000002 1000006 33',
 u'1000002 1000007 8',
 u'1000002 1000009 144',
 u'1000002 1000010 314',
 u'1000002 1000013 8',
 u'1000002 1000014 42',
 u'1000002 1000017 69',
 u'1000002 1000024 329',
 u'1000002 1000025 1']
```

The column index is 2 i.e. the 3rd column

```
rawUserArtistData.map(x => x.split(" ")(2).toDouble).stats()
```

# This will convert the rating to a number

```
rawUserArtistData.map(x => x.split(" ")(2).toDouble).stats()
```

```
(count: 24296858, mean: 15.295762, stdev: 153.915321, max: 439771.000000, min: 1.000000)
```

stats will give you some descriptive measures for the ratings column

```
rawUserArtistData.map(x => x.split(" ")(2).toDouble).stats()
```

```
(count: 24296858, mean: 15.295762, stdev: 153.915321, max: 439771.000000, min: 1.000000)
```

stats operation
only works for
numeric RDDs

```
rawUserArtistData.map(x => x.split(" ")(2).toDouble).stats()
```

```
(count: 24296858, mean: 15.295762, stdev: 153.91532
```

The number of ratings

```
rawUserArtistData.map(x => x.split(" ")(2).toDouble).stats()
```

```
858, mean: 15.295762, stdev: 153.915321, max: 439771.00
```

# Average of the ratings

**Average number** of times a user listens to an artist

```
rawUserArtistData.map(x => x.split(" ")(2).toDouble).stats()
```

```
858, mean: 15.295762, stdev: 153.915321, max: 439771.0
```

Standard deviation of the ratings

```
rawUserArtistData.map(x => x.split(" ")(2).toDouble).stats()
```

```
stdev: 153.915321, max: 439771.000000, min: 1.000000)
```

# Max and min rating

```
rawUserArtistData.map(x => x.split(" ")(2).toDouble).stats()
```

```
(count: 24296858, mean: 15.295762, stdev: 153.915321, max: 439771.000000, min: 1.000000)
```

As you can see this dataset is huge (24 million ratings)

```scala
rawUserArtistData.map(x => x.split(" ")(2).toDouble).stats()
```

```
(count: 24296858, mean: 15.295762, stdev: 153.915321, max: 439771.000000, min: 1.000000)
```

Some of these ratings **might just be noise** - artists that the user listened to very few times

```
rawUserArtistData.map(x => x.split(" ")(2).toDouble).stats()
```

```
(count: 24296858, mean: 15.295762, stdev: 153.915321, max: 439771.000000, min: 1.000000)
```

We'll filter the ratings to include only very strong ratings

```
rawUserArtistData.map(x => x.split(" ")(2).toDouble).stats()
```

```
(count: 24296858, mean: 15.295762, stdev: 153.915321, max: 439771.000000, min: 1.000000)
```

# Also, ALS involves very expensive computations

```
rawUserArtistData.map(x => x.split(" ")(2).toDouble).stats()
```

```
(count: 24296858, mean: 15.295762, stdev: 153.915321, max: 439771.000000, min: 1.000000)
```

If you are running this algorithm on a local machine, filtering low ratings will help

1. Reduce the amount of processing

2. Reduce the amount of data held in-memory

```scala
val rawUserArtistData = sc.textFile(uadatapath)
```

rawUserArtistData is an RDD of strings

```scala
val rawUserArtistData = sc.textFile(uadatapath)
```

Before feeding it to ALS, it should be converted into an RDD of Rating objects

# Rating objects RDD

```
import org.apache.spark.mllib.recommendation._
```

```
val uaData=rawUserArtistData.map(_.split(" ")).filter(_(2).toInt>
```

uaData is an RDD of Rating objects

# Rating objects RDD

```
val uaData=rawUserArtistData.map(_.split(" ")).filter(_(2).toIn
```

Take the RDD of strings

# Rating objects RDD

```
val uaData=rawUserArtistData.map(_.split(" ")).filter(_(2).toIn
```

Split the row into a array

# Rating objects RDD

```
erArtistData.map(_.split(" ")).filter(_(2).toInt>=20).map(x => Ra
```

Filter out any ratings which are below 20

# Rating objects RDD

```
.toInt>=20).map(x => Rating(x(0).toInt,x(1).toInt,x(2).toInt))
```

Convert the array into a Rating object

# Rating objects RDD

```
val uaData=rawUserArtistData.map(_.split(" ")).filter(_(2).toInt>=20).map(x => Rating(x(0).toInt,x(
```

ALS will pass over this RDD many times

# Rating objects RDD

```
val uaData=rawUserArtistData.map(_.split(" ")).filter(_(2).toInt>=20).map(x => Rating(x(0).toInt,x(
```

```
uaData.persist()
```

Persisting will make the computation much faster

# Rating objects RDD

```
val uaData=rawUserArtistData.map(_.split(" ")).filter(_(2).toInt>=20).map(x => Rating(x(0).toInt,x(
```

```
uaData.persist()
```

# Feed uaData to ALS

# ALS

```
val model=ALS.trainImplicit(uaData,10,5,0.01,1)
```

# ALS has 2 methods : train and trainImplicit

# ALS

```
val model=ALS.trainImplicit(uaData,10,5,0.01,1)
```

Since our ratings are implicit ratings, we use the **trainImplicit method**

# ALS

```
val model=ALS.trainImplicit(uaData,10,5,0.01,1)
```

This is the **number of hidden factors** it should look for

# ALS

```
val model=ALS.trainImplicit(uaData,10,5,0.01,1)
```

This is the **max number of iterations** ALS should go through

# ALS

```
val model=ALS.trainImplicit(uaData,10,5,0.01,1)
```

These are lambda and alpha

These parameters are used to control the quality of the ALS results

# ALS

```
val model=ALS.trainImplicit(uaData,10,5,0.01,1)
```

Understanding **how to choose** these 4 parameters

is a topic for another day :)

# ALS

```
val model=ALS.trainImplicit(uaData,10,5,0.01,1)
```

This set of values works pretty well for the Audioscrobbler dataset

# ALS

```
val model=ALS.trainImplicit(uaData,10,5,0.01,1)
```

In general, the choice of parameters is driven by the dataset and the domain

# ALS

```
val model=ALS.trainImplicit(uaData,10,5,0.01,1)
```

There are also separate techniques called hyper-parameter tuning techniques to find the right values

# ALS

```
val model=ALS.trainImplicit(uaData,10,5,0.01,1)
```

```
var recommendations=model.recommendProducts(user,5)
```

The model RDD returned by ALS has a **recommendProducts** method

# ALS

```
val model=ALS.trainImplicit(uaData,10,5,0.01,1)
```

```
var recommendations=model.recommendProducts(user,5)
```

Give this method a user id, and the number of recommendations you want

# ALS

```
val model=ALS.trainImplicit(uaData,10,5,0.01,1)
var recommendations=model.recommendProducts(user,5)
```

This is the beauty of Spark's **MLlib**

# ALS

```
val model=ALS.trainImplicit(uaData,10,5,0.01,1)
var recommendations=model.recommendProducts(user,5)
```

We have implemented the ALS recommendations algorithm in 2 lines of code!

# ALS

```
var recommendations=model.recommendProducts(user,5)
```

```
recommendations
```

```
Array(Rating(1000002,1270,1.185725442054678), Rating(1000002,1000188,1.167724
95057541834), Rating(1000002,1428,1.1576052790420486), Rating(1000002,82,1.13
```

recommendations is an RDD of Rating objects

# ALS

```
var recommendations=model.recommendProducts(user,5)
```

```
recommendations
```

```
Array(Rating(1000002,1270,1.185725442054678), Rating(1000002,1000188,1.167724
95057541834), Rating(1000002,1428,1.1576052790420486), Rating(1000002,82,1.13
```

These are the
recommended Artist Ids

Let's quickly see how good the recommendations are

We'll get recommendations for this user

```
user=1000002
```

# Let's quickly see how good the recommendations are

```
user=1000002
```

In order to assess the quality of recommendations, we'll first see what kind of music this user likes

```
user=1000002
```

```
val userArtists=rawUserArtistData.map(_.split(" ")).filter{case Array(userI
```

We'll find out which artists this user already likes

```
user=1000002
```

```
val userArtists=rawUserArtistData.map(_.split(" ")).filter{case Array(userI
```

The raw user artist data

```
user=1000002
```

```
val userArtists=rawUserArtistData.map(_.split(" ")).filter{case Array
```

Split the row into a list

```
user=1000002
```

```
.filter{case Array(userId,_,rating) => (userId.toInt == user) && (rating.toInt>50)}.ma
```

Filter rows corresponding to this user

```
user=1000002
```

```
.filter{case Array(userId,_,rating) => (userId.toInt == user) && (rating.toInt>50)}.map
```

The user should have listened to these artists at least 50 times

```
user=1000002
```

```
= user) && (rating.toInt>50)}.map(_(1)).collect()
```

# Extract the artist id

It's the second element of the row array

```
user=1000002
```

```
= user) && (rating.toInt>50)}.map(_(1)).collect()
```

# Collect these artists into a list

```
val artistLookup=sc.textFile(artistsPath).map(_.sp
```

Audioscrobbler also provides a file to lookup the artist name for an artist id

```
.textFile(artistsPath).map(_.split("\t")).filter(_.length==2)
```

# Split the row into an array

```
h).map(_.split("\t")).filter(_.length==2).map(x => (x(0),x(1
```

# Make sure the array has 2 elements

```
)).filter(_.length==2).map(x => (x(0),x(1)))
```

Set up each record as a tuple of (Artist ID, Artist Name)

```scala
val artistLookup=sc.textFile(artistsPath).map(_.split("\t")).filter(_.length==2).map(x => (x(0),x(1)))
```

```scala
artistLookup.persist()
```

Persist the RDD as we will be looking it up many times

```
for (artist <- userArtists){
      println( artistLookup.lookup(artist)(0))}
```

Use the lookup action to print the names of the artists this user already likes

```
for (artist <- userArtists){
     println( artistLookup.lookup(artist)(0))}
```

lookup will return an object of type **WrappedArray with 1 element**

```
for (artist <- userArtists){
      println( artistLookup.lookup(artist)(0))}
```

Extract the
**Artist name**
from that array

```
for (artist <- userArtists){
      println( artistLookup.lookup(artist)(0) )}
```

```
Aerosmith
Judas Priest
Metallica
Foo Fighters
Counting Crows
Creed
Audioslave
Muse
(hed) Planet Earth
Dire Straits
Free
Fun Lovin' Criminals
Guns N' Roses
Satriani, Joe
A
Joe Satriani
Bruce Springsteen
```

Looks like this user is a Rock music fan!

```
for (rating <- recommendations){
    println( artistLookup.lookup(rating.product.toString)(0))}
```

```
Queen
Dire Straits
U2
Eric Clapton
Pink Floyd
```

Let's print the recommended Artist names

```
for (rating <- recommendations){
     println( artistLookup.lookup(rating.product.toString)(0))}
```

```
Queen
Dire Straits
U2
Eric Clapton
Pink Floyd
```

**Pretty good recommendations**
**for a Rock music fan!**

Latent Factor analysis and ALS are pretty magical

```
Queen
Dire Straits
U2
Eric Clapton
Pink Floyd
```

We just need to have a good dataset with User-Product Ratings

We just need to have **a good dataset** with User-Product Ratings

Queen
Dire Straits
U2
Eric Clapton
Pink Floyd

The algorithm takes care of finding out **the hidden factors that influence user's preferences**