Music
RECOMMENDATIONS

There are quite a few digital music services available these days

There are quite a few digital music services available these days

Users of these services would be delighted with the huge variety to choose from

But often find it difficult to

Sift through the variety

and identify things they would like

RECOMMENDATIONS HELP USERS

NAVIGATE THE MAZE OF THE MUSIC CATALOGUES

FIND WHAT THEY ARE LOOKING FOR

FIND ARTISTS THEY MIGHT LIKE, BUT DIDN'T KNOW OF

RECOMMENDATIONS HELP USERS
NAVIGATE THE MAZE OF ONLINE STORES
FIND WHAT THEY ARE LOOKING FOR
FIND ARTISTS THEY MIGHT LIKE, BUT DIDN'T KNOW OF

RECOMMENDATIONS HELP THESE SERVICES
SOLVE THE PROBLEM OF DISCOVERY

HOW?

HOW?

USING DATA

WHAT USERS BOUGHT

WHAT USERS BROWSED

WHAT USERS LISTENED TO

WHAT USERS RATED

→ **RECOMMENDATION ENGINE** →

TOP PICKS FOR YOU!!

IF YOU LIKE THIS, YOU'LL LOVE THAT!

RECOMMENDATION ENGINE

OBJECTIVE

FILTER RELEVANT PRODUCTS

PREDICT WHAT RATING THE USER WOULD GIVE A PRODUCT

PREDICT WHETHER A USER WOULD BUY A PRODUCT

RANK PRODUCTS BASED ON THEIR RELEVANCE TO THE USER

TASKS PERFORMED BY RECOMMENDATION ENGINES

MOST RECOMMENDATION ENGINES USE A TECHNIQUE CALLED

COLLABORATIVE FILTERING

COLLABORATIVE FILTERING

HOW DOES THAT WORK?

HOW DO YOU NORMALLY FIND

A MOVIE TO WATCH?

A RESTAURANT TO GO TO?

AN ARTIST TO CHECK OUT?

A BOOK TO READ?

SOMEONE WHO LIKES

[redacted]

COLLABORATIVE FILTERING

HOW DOES THAT WORK?

THE BASIC PREMISE IS THAT

IF 2 USERS HAVE THE SAME OPINION
ABOUT A BUNCH OF PRODUCTS

THEY ARE LIKELY TO HAVE THE SAME
OPINION ABOUT OTHER PRODUCTS TOO

COLLABORATIVE FILTERING IS A GENERAL TERM

FOR ANY ALGORITHM THAT RELIES ONLY ON USER BEHAVIOR (HISTORY, RATINGS, SIMILAR USERS ETC)

COLLABORATIVE FILTERING IS A GENERAL TERM

FOR ANY ALGORITHM THAT RELIES ONLY ON USER BEHAVIOR
(HISTORY, RATINGS, SIMILAR USERS ETC)

THIS IS AS OPPOSED TO CONTENT BASED FILTERING (WHICH USES PRODUCT ATTRIBUTES LIKE GENRE, DESCRIPTION ETC)

COLLABORATIVE FILTERING IS A GENERAL TERM

FOR ANY ALGORITHM THAT RELIES ONLY ON USER BEHAVIOR (HISTORY, RATINGS, SIMILAR USERS ETC)

CF ALGORITHMS NORMALLY PREDICT USERS' RATINGS FOR PRODUCTS THEY HAVEN'T YET RATED

RATING HERE IS A GENERAL TERM

IT CAN BE A RATING THE USER HAS EXPLICITLY GIVEN

EXPLICIT RATING

NETFLIX ASKS USERS TO RATE A MOVIE ONCE THEY HAVE WATCHED IT

IT CAN BE BASED ON A PREFERENCE THE USER HAS SOMEHOW INDICATED

IMPLICIT RATING

TIMES THE USER LISTENED TO AN ARTIST

THERE ARE MANY MANY DIFFERENT ALGORITHMS TO PERFORM COLLABORATIVE FILTERING

1 POPULAR TECHNIQUE IS

LATENT FACTOR ANALYSIS

LATENT FACTOR ANALYSIS

IDENTIFY HIDDEN FACTORS THAT INFLUENCE A USER'S RATING

SOMETIMES THE FACTORS MIGHT TURN OUT TO HAVE MEANING (LIKE GENRE OR OVERALL POPULARITY)

OTHER TIMES, THEY MIGHT BE ABSTRACT FACTORS WITH NO REAL LIFE MEANING

LATENT FACTOR ANALYSIS

LET'S SAY YOU WERE DOING THIS WITH MOVIES

AND THE HIDDEN FACTORS ARE

1. COMMERCIAL APPEAL
2. DRAMATIC VS COMEDIC NATURE

MOVIES ARE REPRESENTED USING THESE DESCRIPTORS

COMMERCIAL

USERS ARE REPRESENTED USING THE SAME DESCRIPTORS

JANANI WATCHED 10 MOVIES
7 COMMERCIAL, 9 DRAMA

DRAMA

COMEDY

ARTY

RECOMMENDATIONS FOR JANANI

COMMERCIAL

COMEDY

DRAMA

ARTY

LATENT FACTOR COLLABORATIVE FILTERING

TO IDENTIFY HIDDEN FACTORS

YOU NEED A USER-PRODUCT-RATING MATRIX

	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	

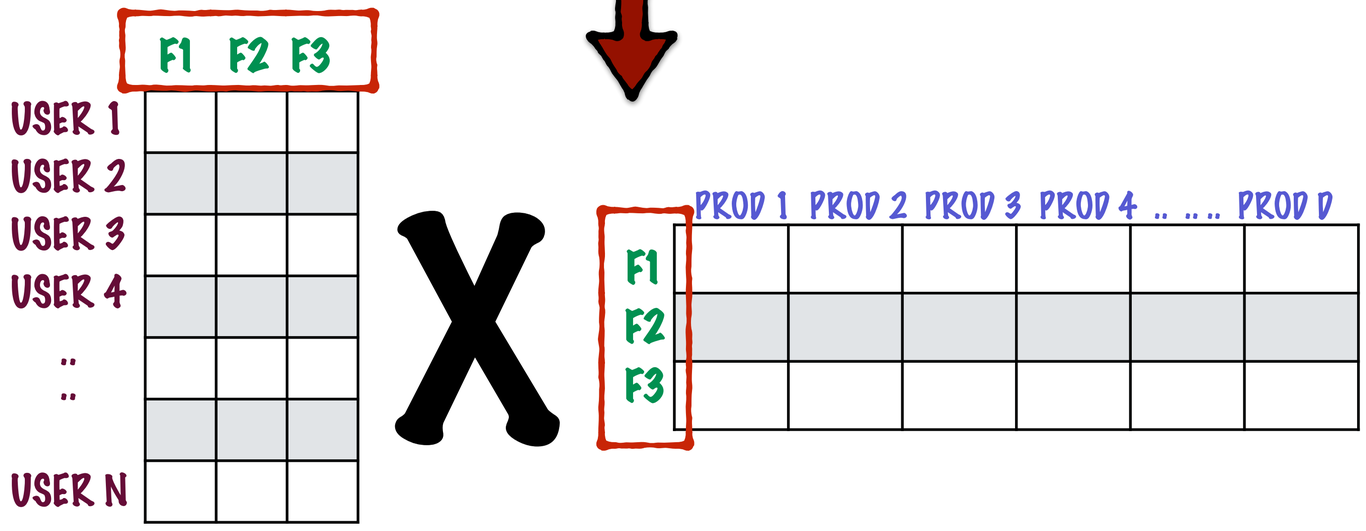
IT REPRESENTS USERS BY THEIR RATINGS FOR DIFFERENT PRODUCTS

LATENT FACTOR COLLABORATIVE FILTERING

IT REPRESENTS USERS BY THEIR RATINGS FOR DIFFERENT PRODUCTS

	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

BREAK THIS DOWN TO IDENTIFY THE HIDDEN FACTORS



LATENT FACTOR COLLABORATIVE FILTERING

R

USER-PROD RATING MATRIX

	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

DECOMPOSED INTO

P

	F1	F2	F3
USER 1			
USER 2			
USER 3			
USER 4			
.. ..			
USER N			

USER-FACTOR MATRIX

EACH ROW IS A USER DESCRIBED BY THEIR INTEREST IN THE HIDDEN FACTORS

X

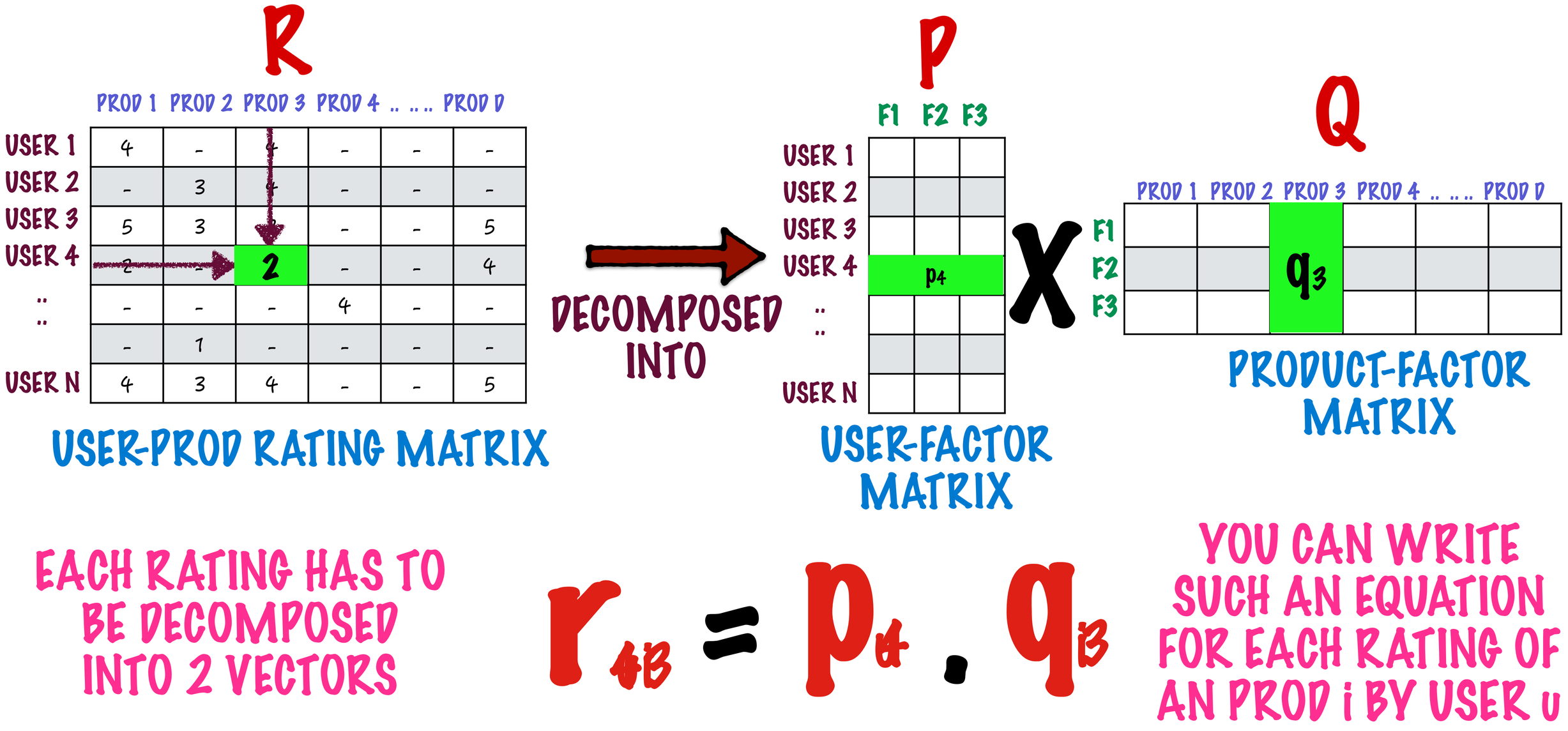
Q

	PROD 1	PROD 2	PROD 3	PROD 4	.. .. ..	PROD D
F1						
F2						
F3						

PRODUCT-FACTOR MATRIX

EACH COLUMN IS A PRODUCT DESCRIBED BY ITS RELEVANCE TO THE HIDDEN FACTORS

LATENT FACTOR COLLABORATIVE FILTERING



EACH RATING HAS TO BE DECOMPOSED INTO 2 VECTORS

$$r_{4B} = p_4 \cdot q_B$$

YOU CAN WRITE SUCH AN EQUATION FOR EACH RATING OF AN PROD i BY USER u

LATENT FACTOR COLLABORATIVE FILTERING

THIS WILL GIVE US VALUES FOR P1, P2 ETC AS WELL AS Q1, Q2 ETC

R

	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

USER-PROD RATING MATRIX

DECOMPOSED INTO

P

	F1	F2	F3
USER 1			
USER 2	p_2		
USER 3			
USER 4			
.. ..			
USER N			

USER-FACTOR MATRIX

X

Q

	PROD 1	PROD 2	PROD 3	PROD 4	.. .. ..	PROD D
F1						
F2				q_4		
F3						

PRODUCT-FACTOR MATRIX

SOLVE THIS SET OF EQUATIONS FOR THE SET OF RATINGS WHICH EXIST

$$r_{ui} = p_u \cdot q_i$$

USE THE RESULTING P'S AND Q'S TO FIND THE RATING OF ANY USER FOR ANY PRODUCT

LATENT FACTOR COLLABORATIVE FILTERING

SOLVE THIS SET OF EQUATIONS FOR THE SET OF RATINGS WHICH EXIST (TRAINING SET)

$$r_{ui} = p_u . q_i$$

ONCE WE'VE SOLVED THIS SET OF EQUATIONS WE CAN PREDICT WHAT RATING THE USER WOULD GIVE TO ANY PRODUCT

LATENT FACTOR COLLABORATIVE FILTERING

SOLVE THIS SET OF EQUATIONS FOR THE SET OF RATINGS WHICH EXIST (TRAINING SET)

$$r_{ui} = p_u . q_i$$

SORT THE PREDICTED RATINGS IN DESCENDING ORDER TO FIND THE TOP RECOMMENDATIONS FOR A USER

LATENT FACTOR COLLABORATIVE FILTERING

SOLVE THIS SET OF EQUATIONS FOR THE SET OF RATINGS WHICH EXIST (TRAINING SET)

$$r_{ui} = p_u . q_i$$

ALTERNATING LEAST SQUARES

IS A TECHNIQUE TO SOLVE THIS SET OF EQUATIONS

ALTERNATING LEAST SQUARES

ALS IS A STANDARD OPTIMIZATION TECHNIQUE THAT CAN BE APPLIED TO MANY PROBLEMS

SPARK'S MLLIB HAS A BUILT-IN CLASS FOR APPLYING ALS ON ANY USER-PRODUCT-RATING MATRIX

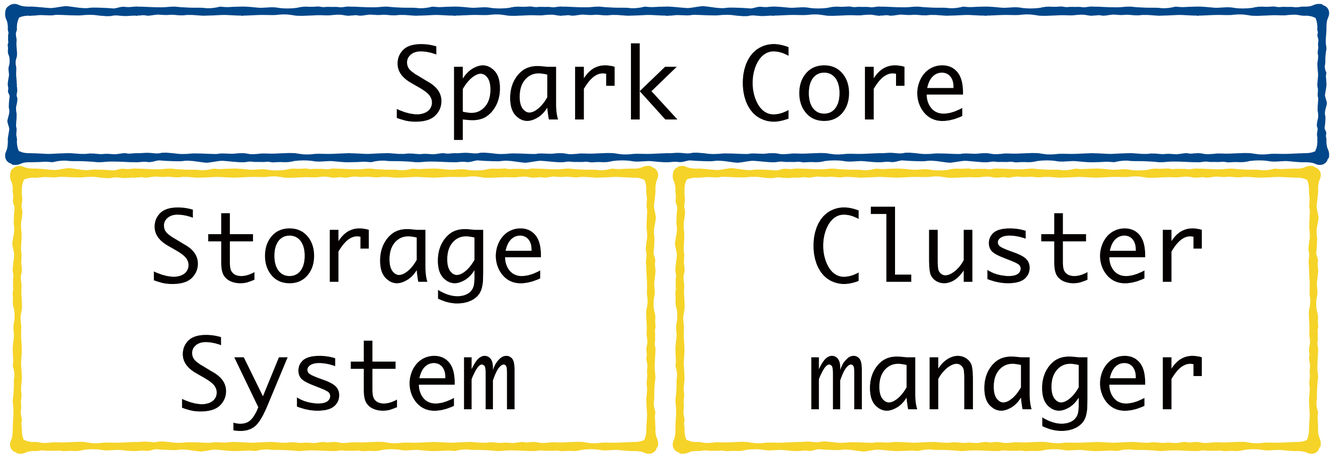
ALTERNATING LEAST SQUARES

MLLIB COMPLETELY ABSTRACTS AWAY THE TECHNICAL IMPLEMENTATION DETAILS OF ALS

Recap

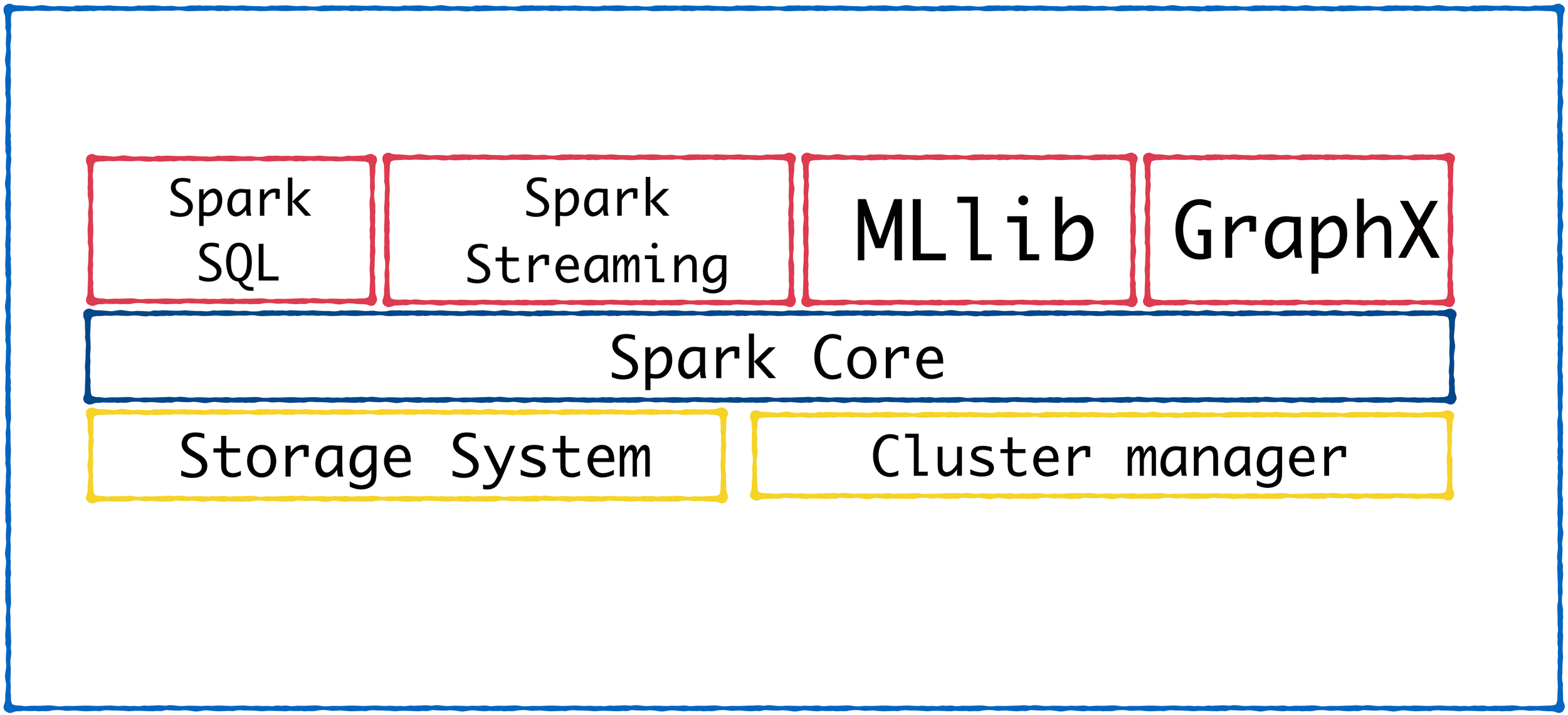
APACHE SPARK

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



Recap

APACHE SPARK



Recap

APACHE SPARK

MLlib provides built-in Machine Learning functionality in Spark

Recap

APACHE SPARK

This basically solves 2 problems with previous computing frameworks

Abstraction

Performance

Recap

APACHE SPARK

Abstraction

MLlib

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

These libraries are not suited for distributed computing

Recap

APACHE SPARK

Abstraction

MLlib

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

Abstraction

MLlib

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

Recap

APACHE SPARK

Abstraction

MLlib

This completely abstracts the programmer from

Implementing the ML algorithm

Intricacies of running it across a cluster

Recap

APACHE SPARK

This basically solves 2 problems with previous computing frameworks

MLlib

Abstraction

Performance

Recap

APACHE SPARK

Performance

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

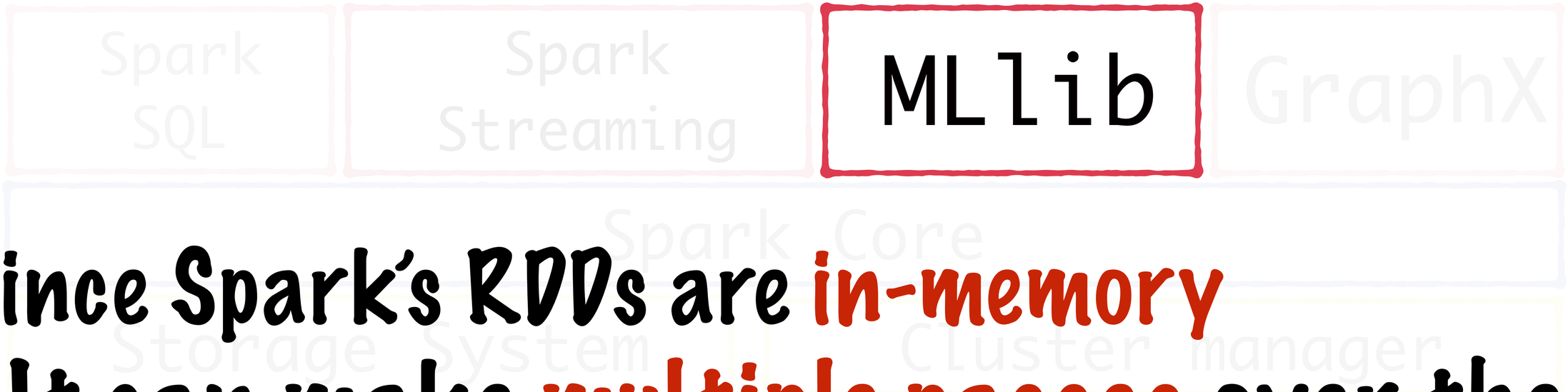
MLlib

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

Recap

APACHE SPARK

Performance



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!

Audioscrobbler Dataset

MLlib

Let's now look at the code

Load the dataset with User-Artist ratings

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

```
rawUserArtistData.take(10)
```

```
[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']
```

```
rawUserArtistData.take(10)
```

```
[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']
```

User ID

```
rawUserArtistData.take(10)
```

```
[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']
```

Artist ID

```
rawUserArtistData.take(10)
```

```
[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']
```

times user listened to the artist

Implicit rating

```
rawUserArtistData.take(10)
```

```
[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']
```

Let's get a quick sense of this ratings column

```
rawUserArtistData.map(lambda x:float(x.split(" ")[2])).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']
```

Extract the ratings column

```
rawUserArtistData.map(lambda x:float(x.split(" ")[2])).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(lambda x:float x.split(" ")[2])).stats()
```

This will convert the rating to a number

```
rawUserArtistData.map(lambda x:float(x.split(" ")[2])).stats()
```

```
(count: 24296858, mean: 15.2957624809, stdev: 153.915321302, max: 439771.0, min: 1.0)
```

stats will give you some descriptive measures for the ratings column

```
rawUserArtistData.map(lambda x:float(x.split(" ")[2])).stats()
```

```
(count: 24296858, mean: 15.2957624809, stdev: 153.915321302, max: 439771.0, min: 1.0)
```

stats operation

only works for numeric RDDs

```
rawUserArtistData.map(lambda x:float(x.split(" ")[2])).stats()
```

```
(count: 24296858, mean: 15.2957624809, stdev: 15
```

The number of ratings

```
rawUserArtistData.map(lambda x:float(x.split(" ")[2])).stats()
```

```
: 24296858, mean: 15.2957624809, stdev: 153.915321
```

Average of the ratings

Average number of times a user listens to an artist

```
rawUserArtistData.map(lambda x:float(x.split(" ")[2])).stats()
```

```
n: 15.2957624809, stdev: 153.915321302 max: 439771.0, min: 1.0)
```

Standard deviation of the ratings

```
rawUserArtistData.map(lambda x:float(x.split(" ")[2])).stats()
```

```
4809, stdev: 153.915321302, max: 439771.0, min: 1.0)
```

Max and min rating

```
rawUserArtistData.map(lambda x:float(x.split(" ")[2])).stats()
```

```
(count: 24296858, mean: 15.2957624809, stdev: 153.915321302, max: 439771.0, min: 1.0)
```

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

```
rawUserArtistData.map(lambda x:float(x.split(" ")[2])).stats()
```

```
(count: 24296858, mean: 15.2957624809, stdev: 153.915321302, max: 439771.0, min: 1.0)
```

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

```
rawUserArtistData.map(lambda x:float(x.split(" ")[2])).stats()
```

```
(count: 24296858, mean: 15.2957624809, stdev: 153.915321302, max: 439771.0, min: 1.0)
```

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

```
rawUserArtistData.map(lambda x:float(x.split(" ")[2])).stats()
```

```
(count: 24296858, mean: 15.2957624809, stdev: 153.915321302, max: 439771.0, min: 1.0)
```

Also, ALS involves very expensive computations

```
rawUserArtistData.map(lambda x:float(x.split(" ")[2])).stats()
```

```
(count: 24296858, mean: 15.2957624809, stdev: 153.915321302, max: 439771.0, min: 1.0)
```

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

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

rawUserArtistData is an RDD of strings

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

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

Rating objects RDD

```
from pyspark.mllib.recommendation import Rating,ALS
```

```
uaData=rawUserArtistData\
     .map(lambda x:x.split(" "))\
     .filter(lambda x: float(x[2])>=20)\
     .map(lambda x:Rating(x[0],x[1],x[2]))
uaData.persist()
```

Rating objects RDD

```
uaData=rawUserArtistData\
      .map(lambda x:x.split(" "))\
      .filter(lambda x: float(x[2])>=20)\
      .map(lambda x:Rating(x[0],x[1],x[2]))
uaData.persist()
```

Take the RDD of strings

Rating objects RDD

```
uaData=rawUserArtistData\
      .map(lambda x:x.split(" "))\
      .filter(lambda x: float(x[2])>=20)\
      .map(lambda x:Rating(x[0],x[1],x[2]))
uaData.persist()
```

Split the row into a list

Rating objects RDD

```
uaData=rawUserArtistData\
      .map(lambda x:x.split(" "))\
      .filter(lambda x: float(x[2])>=20)\
      .map(lambda x:Rating(x[0],x[1],x[2]))
uaData.persist()
```

Filter out any ratings which are below 20

Rating objects RDD

```
uaData=rawUserArtistData\
      .map(lambda x:x.split(" "))\
      .filter(lambda x: float(x[2])>=20)\
      .map(lambda x:Rating(x[0],x[1],x[2]))
uaData.persist()
```

Convert the list into a Rating object

Rating objects RDD

```
uaData=rawUserArtistData\
     .map(lambda x:x.split(" "))\
     .filter(lambda x: float(x[2])>=20)\
     .map(lambda x:Rating(x[0],x[1],x[2]))
uaData.persist()
```

ALS will pass over this RDD many times

Rating objects RDD

```
uaData=rawUserArtistData\
      .map(lambda x:x.split(" "))\
      .filter(lambda x: float(x[2])>=20)\
      .map(lambda x:Rating(x[0],x[1],x[2]))
uaData.persist()
```

Persisting will make the computation much faster

Rating objects RDD

```
uaData=rawUserArtistData\
    .map(lambda x:x.split(" "))\
    .filter(lambda x: float(x[2])>=20)\
    .map(lambda x:Rating(x[0],x[1],x[2]))
uaData.persist()
```

```
uaData.take(10)
```

```
[Rating(user=1000002, product=1, rating=55.0),
 Rating(user=1000002, product=1000006, rating=33.0),
 Rating(user=1000002, product=1000009, rating=144.0),
 Rating(user=1000002, product=1000010, rating=314.0),
 Rating(user=1000002, product=1000014, rating=42.0),
 Rating(user=1000002, product=1000017, rating=69.0),
 Rating(user=1000002, product=1000024, rating=329.0),
 Rating(user=1000002, product=1000031, rating=47.0),
 Rating(user=1000002, product=1000055, rating=25.0),
 Rating(user=1000002, product=1000062, rating=71.0)]
```

Rating objects RDD

```
uaData.take(10)
```

```
[Rating(user=1000002, product=1, rating=55.0),
 Rating(user=1000002, product=1000006, rating=33.0),
 Rating(user=1000002, product=1000009, rating=144.0),
 Rating(user=1000002, product=1000010, rating=314.0),
 Rating(user=1000002, product=1000014, rating=42.0),
 Rating(user=1000002, product=1000017, rating=69.0),
 Rating(user=1000002, product=1000024, rating=329.0),
 Rating(user=1000002, product=1000031, rating=47.0),
 Rating(user=1000002, product=1000055, rating=25.0),
 Rating(user=1000002, product=1000062, rating=71.0)]
```

Feed uaData to ALS

ALS

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

ALS has 2 methods :
train and trainImplicit

ALS

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

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

ALS

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

This is the number of hidden factors it should look for

ALS

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

This is the max number of iterations ALS should go through

ALS

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

This is lambda

A parameter used to control the quality of the ALS results

ALS

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

Understanding how to choose these 3 parameters is a topic for another day :)

ALS

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

This set of values works pretty well for the Audioscrobbler dataset

ALS

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

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

ALS

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

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

ALS

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

The model RDD returned by ALS has a recommendProducts method

ALS

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

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

ALS

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

This is the beauty of Spark's MLlib

ALS

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

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

ALS

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

```
recommendations
```

```
[Rating(user=1000002, product=1000113, rating=0.8888236136225758),
 Rating(user=1000002, product=1205, rating=0.8841759358310398),
 Rating(user=1000002, product=82, rating=0.8602128285583197),
 Rating(user=1000002, product=1274, rating=0.855939103947302 8),
 Rating(user=1000002, product=976, rating=0.7868346783167373)]
```

recommendations is an RDD of Rating objects

ALS

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

```
recommendations
```

```
[Rating(user=1000002, product=1000113, rating=0.8888236136225758),
 Rating(user=1000002, product=1205, rating=0.8841759358310398),
 Rating(user=1000002, product=82, rating=0.8602128285583197),
 Rating(user=1000002, product=1274, rating=0.855939103947302 8),
 Rating(user=1000002, product=976, rating=0.786834678316737 3)]
```

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
```

```
userArtists=rawUserArtistData\
    .map(lambda x:x.split(" "))\
    .filter(lambda x:int(x[0])==user and int(x[2])>50)\
    .map(lambda x:x[1]).collect()
```

We'll find out which artists this user already likes

```
user=1000002
```

```
userArists=rawUserArtistData\
    .map(lambda x:x.split(" "))\
    .filter(lambda x:int(x[0])==user and int(x[2])>50)\
    .map(lambda x:x[1]).collect()
```

The raw user artist data

```
user=1000002
```

```
userArtists=rawUserArtistData\
    .map(lambda x:x.split(" "))\
    .filter(lambda x:int(x[0])==user and int(x[2])>50)\
    .map(lambda x:x[1]).collect()
```

Split the row into a list

```
user=1000002
```

```
userArtists=rawUserArtistData\
    .map(lambda x:x.split(" "))\
    .filter(lambda x:int(x[0])==user and int(x[2])>50)\
    .map(lambda x:x[1]).collect()
```

Filter rows corresponding to this user

```
user=1000002
```

```
userArtists=rawUserArtistData\
    .map(lambda x:x.split(" "))\
    .filter(lambda x:int(x[0])==user and int(x[2])>50)\
    .map(lambda x:x[1]).collect()
```

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

```
user=1000002
```

```
userArtists=rawUserArtistData\
    .map(lambda x:x.split(" "))\
    .filter(lambda x:int(x[0])==user and int(x[2])>50)\
    .map(lambda x:x[1]).collect()
```

Extract the artist id

```
user=1000002
```

```
userArtists=rawUserArtistData\
    .map(lambda x:x.split(" "))\
    .filter(lambda x:int(x[0])==user and int(x[2])>50)\
    .map(lambda x:x[1]).collect()
```

Collect these artists into a list

```
artistLookup=sc.textFile(artistsPath).map(lambda x:x.split("\t"))
artistLookup.persist()
```

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

```
artistLookup=sc.textFile(artistsPath).map(lambda x:x.split("\t"))
artistLookup.persist()
```

Load this into a text file

```
artistLookup=sc.textFile(artistsPath).map(lambda x:x.split("\t"))
artistLookup.persist()
```

Split the row into a tuple of (Artist ID, Artist Name)

```
artistLookup=sc.textFile(artistsPath).map(lambda x:x.split("\t"))
artistLookup.persist()
```

Persist it as we will be looking it up many times

```
artistLookup=sc.textFile(artistsPath).map(lambda x:x.split("\t"))
artistLookup.persist()
```

```
for artist in userArtists:
    print artistLookup.lookup(artist)
```

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

```
artistLookup=sc.textFile(artistsPath).map(lambda x:x.split("\t"))
artistLookup.persist()
```

```
for artist in userArtists:
    print artistLookup.lookup(artist)
```

```
[u'Portishead']
[u'Phil Collins Big Band']
[u'The Phil Collins Big Band']
[u'A Perfect Circle']
[u'Aerosmith']
[u'MC Hawking']
[u'Pantera']
[u'Judas Priest']
[u'Metallica']
[u'Terrorvision']
[u'Lynyrd Skynyrd']
[u'3 Doors Down']
```

Looks like this user is a Rock music fan!

```
for rating in recommendations:
    print artistLookup.lookup(str(rating.product))
```

```
[u'The Beatles']
[u'U2']
[u'Pink Floyd']
[u'Red Hot Chili Peppers']
[u'Nirvana']
```

Let's print the recommended Artist names

```
for rating in recommendations:
    print artistLookup.lookup(str(rating.product))
```

```
[u'The Beatles']
[u'U2']
[u'Pink Floyd']
[u'Red Hot Chili Peppers']
[u'Nirvana']
```

Pretty good recommendations
for a Rock music fan!

Latent Factor analysis and ALS are pretty magical

```
[u'The Beatles']
[u'U2']
[u'Pink Floyd']
[u'Red Hot Chili Peppers']
[u'Nirvana']
```

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

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

```
[u'The Beatles']
[u'U2']
[u'Pink Floyd']
[u'Red Hot Chili Peppers']
[u'Nirvana']
```

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