

Recommender System with *Hadoop and Spark*

Hadoop

- Basically parallel processing framework running on cluster of commodity machines
- Stateless functional programming because processing of each row of data does not depend upon any other row or state
- Data is replicated, partitioned and resides in Hadoop Distributed File System (HDFS)
- Each partitions get processed by a separated mapper or reducer task

Item-based Collaborative Filtering

- the similarities between different items in the dataset are calculated according to the items' properties
- these similarity values are used to recommend items to users
- One problems?

If a lot of users like item A and B, but A and B have no similarity such as beer and baby diapers?

- Theoretical Background
- There are generally two basic algorithms for the recommender system, user collaborative filtering (user CF) and item collaborative filtering (item CF). User CF is a form of collaborative filtering based on the similarity between users calculated using people's ratings of those items while item CF is based on the similarity between items.
- In this assignment, we use item CF instead of user CF for the following reasons:
 - The number of users weighs more than the number of products, which makes item CF more computationally efficient.
 - Items (movie) will not change frequently in this project, while users' preferences may change frequently. Item CF can help save computation.
 - Recommending movies based on users' historical data makes more sense intuitively.

- To build the item CF model we first need to define the relationship between items. Some possible strategies are making use of the categories, producers, actors, countries, years, earnings and so on. Here, however, we use rating history to build relationships between movies based on limited data, which means the number of users rated two movies denotes the level of how these two movies are connected.

- A co-occurrence matrix is a matrix that is defined over an image to be the distribution of co-occurring pixel values (grayscale values, or colors) at a given offset. In this project co-occurrence matrix denotes the frequency any two movies show up together. Then we normalize the co-occurrence matrix and multiply it with the user rating matrix to get the final result, as shown in the figure below.

Item-Based Recommender using Co-occurrence similarity

1,101,5.0
1,102,3.0
1,103,2.5
2,101,2.0
2,102,2.5
2,103,5.0
2,104,2.0
3,101,2.0
3,104,4.0
3,105,4.5
3,107,5.0
4,101,5.0
4,103,3.0
4,104,4.5
4,106,4.0
5,101,4.0
5,102,3.0
5,103,2.0
5,104,4.0
5,105,3.5
5,106,4.0

Look at User 1 and 2 who have watched Movie 101, you will see that there is 2 cooccurrence of Movie 102



	[101]	[102]	[103]	[104]	[105]	[106]	[107]
[101]	5	3	4	4	2	2	1
[102]	3	3	3	2	1	1	0
[103]	4	3	4	3	1	2	0
[104]	4	2	3	4	2	2	1
[105]	2	1	1	2	2	1	1
[106]	2	1	2	2	1	2	0
[107]	1	0	0	1	1	0	1

	101	102	103	104	105	106	107		U3		R
101	5	3	4	4	2	2	1	X	2.0	=	40.0
102	3	3	3	2	1	1	0		0.0		18.5
103	4	3	4	3	1	2	0		0.0		24.5
104	4	2	3	4	2	2	1		4.0		40.0
105	2	1	1	2	2	1	1		4.5		26.0
106	2	1	2	2	1	2	0		0.0		16.5
107	1	0	0	1	1	0	1		5.0		15.5

Implement of the recommender system

- Built history matrix — contains the interactions between users and items as a user-by-item matrix
- Built co-occurrence matrix — transforms the history matrix into an item-by-item matrix, recording which items appear together in user histories
- Do recommendation

HISTORY MATRIX

1,101,5.0
1,102,3.0
1,103,2.5
2,101,2.0
2,102,2.5
2,103,5.0
2,104,2.0
3,101,2.0
3,104,4.0
3,105,4.5
3,107,5.0
4,101,5.0
4,103,3.0
4,104,4.5
4,106,4.0
5,101,4.0
5,102,3.0
5,103,2.0
5,104,4.0
5,105,3.5
5,106,4.0



Map (key, value)
key.set (User ID)
Value.set (ItemID+
":" + Rating)

<1, (101: 5.0)>
<1, (102:3.0)>



Combine(key,
list(value))

<1, ((101: 5.0),
(102:3.0))>



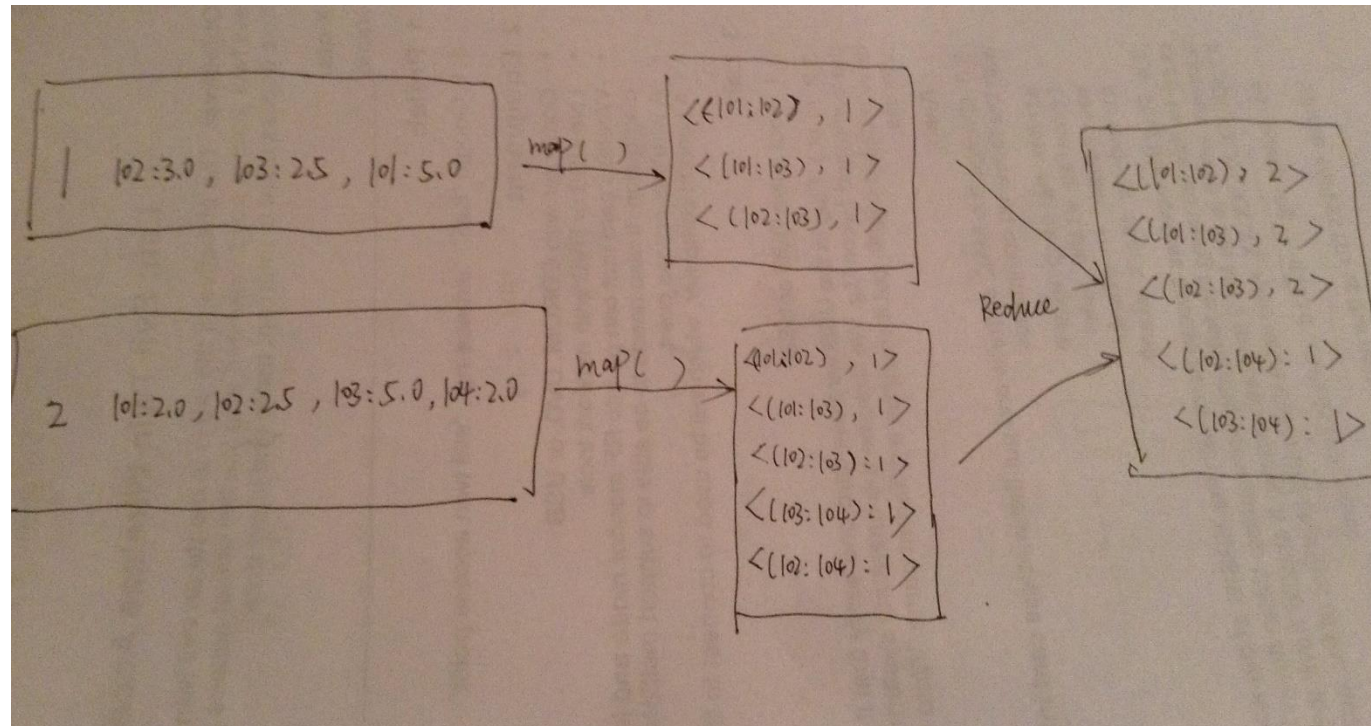
Reduce



1	102:3.0,103:2.5,101:5.0
2	101:2.0,102:2.5,103:5.0,104:2.0
3	107:5.0,101:2.0,104:4.0,105:4.5
4	101:5.0,103:3.0,104:4.5,106:4.0
5	101:4.0,102:3.0,103:2.0,104:4.0,105:3.5,106:4.0

Co-occurrence matrix

- transforms the history matrix into an item-by-item matrix, recording which items appear together in user histories



```
101:101 5
101:102 3
101:103 4
101:104 4
101:105 2
101:106 2
101:107 1
102:101 3
102:102 3
102:103 3
102:104 2
102:105 1
102:106 1
103:101 4
103:102 3
103:103 4
103:104 3
103:105 1
```

Indicator Matrix

101:101	5
101:102	3
101:103	4
101:104	4
101:105	2
101:106	2
101:107	1
102:101	3
102:102	3
102:103	3
102:104	2
102:105	1
102:106	1
103:101	4
103:102	3
103:103	4
103:104	3
103:105	1



U3	
[101]	2.0
[102]	0.0
[103]	0.0
[104]	4.0
[105]	4.5
[106]	0.0
[107]	5.0



3	107,15.5
3	106,16.5
3	105,26.0
3	104,38.0
3	103,24.5
3	102,18.5
3	101,40.0

Validation

- Select 20% users from the data set, and assume 20% movies have not been watched

	101	102	103	104	105	106	107		U3		R
101	5	3	4	4	2	2	1		2.0		40.0
102	3	3	3	2	1	1	0		0.0		18.5
103	4	3	4	3	1	2	0	X	0.0	=	24.5
104	4	2	3	4	2	2	1		4.0		40.0
105	2	1	1	2	2	1	1		4.5		26.0
106	2	1	2	2	1	2	0		0.0		16.5
107	1	0	0	1	1	0	1		5.0		15.5

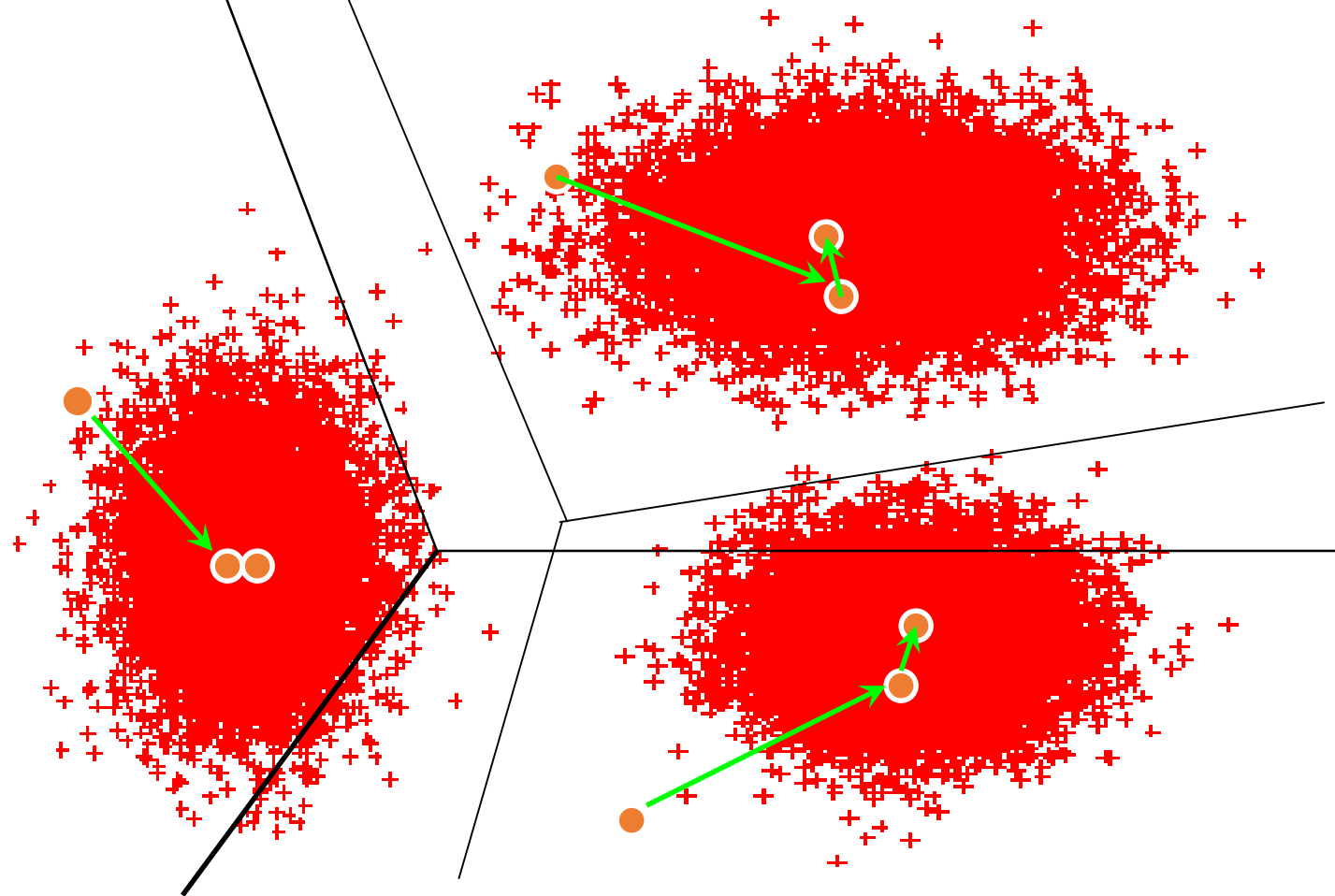
Machine Learning with MapReduce

Algorithm 16.1: K-means Algorithm

K-means (D, k, ϵ) :

```
1  $t = 0$ 
2 Randomly initialize  $k$  centroids:  $\mu_1^t, \mu_2^t, \dots, \mu_k^t$ 
3 repeat
4    $t = t + 1$ 
5   // Cluster Assignment Step
6   foreach  $x_j \in D$  do
7      $j^* = \arg \min_i \{\|x_j - \mu_i^t\|^2\}$  // Assign  $x_j$  to closest centroid
8      $C_{j^*} = C_{j^*} \cup \{x_j\}$ 
9   // Centroid Update Step
10  foreach  $i = 1$  to  $k$  do
11     $\mu_i^t = \frac{1}{|C_i|} \sum_{x_j \in C_i} x_j$ 
12 until  $\sum_{i=1}^k \|\mu_i^t - \mu_i^{t-1}\|^2 \leq \epsilon$ 
```

K-Means Clustering



How to MapReduce K-Means?

- Given K , assign the first K random points to be the initial cluster centers
- Assign subsequent points to the closest cluster using the supplied distance measure
- Compute the centroid of each cluster and iterate the previous step until the cluster centers converge within *delta*
- Run a final pass over the points to cluster them for output

K-Means Map/Reduce Design

- Driver
 - Runs multiple iteration jobs using mapper+combiner+reducer
 - Runs final clustering job using only mapper
- Mapper
 - Configure: Single file containing encoded Clusters
 - Input: File split containing encoded Vectors
 - Output: Vectors keyed by nearest cluster
- Combiner
 - Input: Vectors keyed by nearest cluster
 - Output: Cluster centroid vectors keyed by “cluster”
- Reducer (singleton)
 - Input: Cluster centroid vectors
 - Output: Single file containing Vectors keyed by cluster

Mapper - mapper has k centers in memory.

Input Key-value pair (each input data point x).

Find the index of the closest of the k centers (call it iClosest).

Emit: (key,value) = (iClosest, x)

Reducer(s) – Input (key,value)

Key = index of center

Value = iterator over input data points closest to ith center

At each key value, run through the iterator and average all the
Corresponding input data points.

Emit: (index of center, new center)

Improved Version: Calculate partial sums in mappers

Mapper - mapper has k centers in memory. Running through one input data point at a time (call it x). Find the index of the closest of the k centers (call it iClosest). Accumulate sum of inputs segregated into K groups depending on which center is closest.

Emit: (, partial sum)

Or

Emit(index, partial sum)

Reducer – accumulate partial sums and

Emit with index or without

Mahout

- Mahout is a machine-learning library with tools for [clustering, classification, and several types of recommenders](#), including tools to calculate most-similar items or build item recommendations for users. Mahout employs the Hadoop framework to distribute calculations across a cluster
- Command
 - “mahout recommenditembased --input /movielens.txt
 - output recommendations --numRecommendations 10
 - outputPathForSimilarityMatrix similarity-matrix
 - similarityClassname SIMILARITY_COOCCURRENCE”

User-based CF RS with Spark

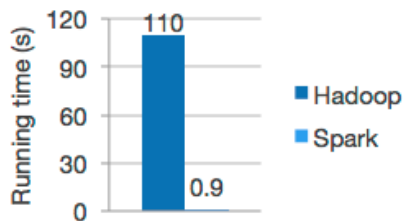
- Task: to implement a recommendation system using *Apache Spark* framework
- Approach:
 - user-based collaborative filtering
 - Similarity metric
- Dataset: MovieLens

Spark

Speed

Run programs up to 100x faster than Hadoop MapReduce in memory, or 10x faster on disk.

Spark has an advanced DAG execution engine that supports cyclic data flow and in-memory computing.



Logistic regression in Hadoop and Spark

Ease of Use

Write applications quickly in Java, Scala or Python.

Spark offers over 80 high-level operators that make it easy to build parallel apps. And you can use it *interactively* from the Scala and Python shells.

```
file = spark.textFile("hdfs://...")

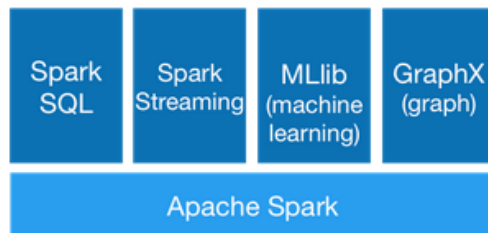
file.flatMap(lambda line: line.split())
      .map(lambda word: (word, 1))
      .reduceByKey(lambda a, b: a+b)
```

Word count in Spark's Python API

Generality

Combine SQL, streaming, and complex analytics.

Spark powers a stack of high-level tools including [Spark SQL](#), [MLlib](#) for machine learning, [GraphX](#), and [Spark Streaming](#). You can combine these libraries seamlessly in the same application.



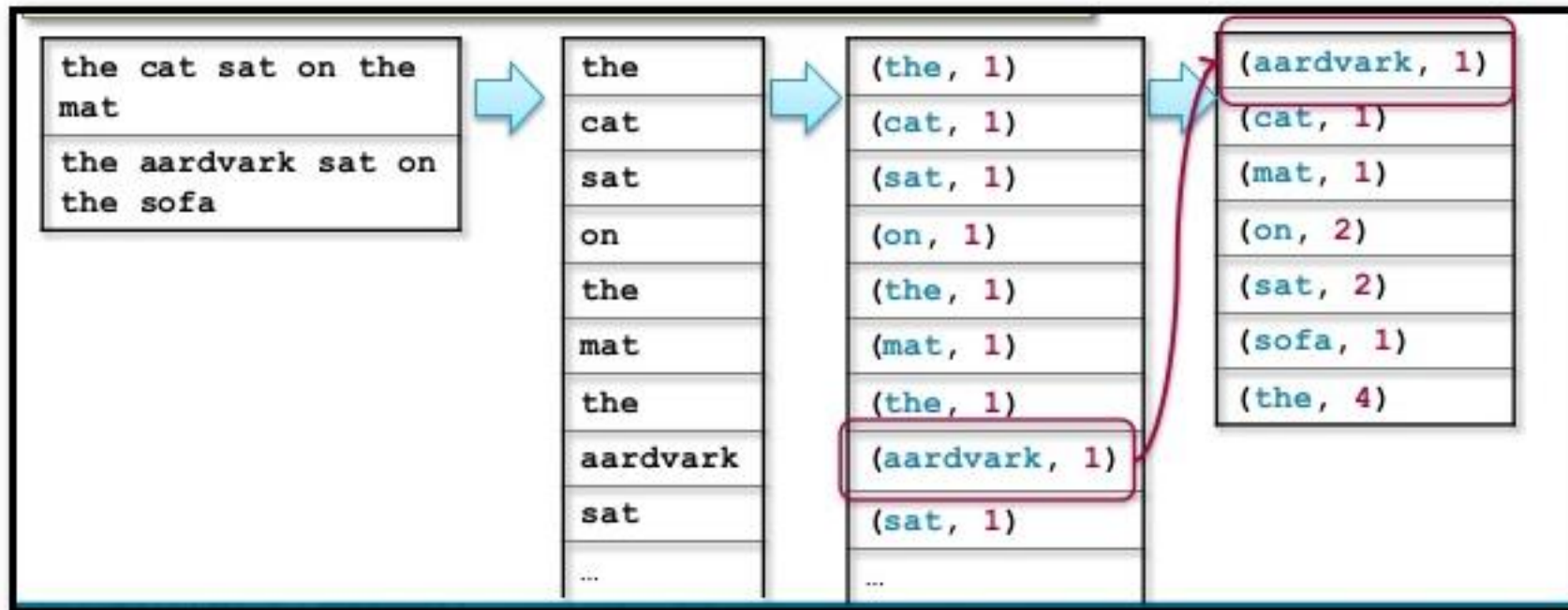
Spark operators

Transformations	$\text{map}(f : T \Rightarrow U) : \text{RDD}[T] \Rightarrow \text{RDD}[U]$ $\text{filter}(f : T \Rightarrow \text{Bool}) : \text{RDD}[T] \Rightarrow \text{RDD}[T]$ $\text{flatMap}(f : T \Rightarrow \text{Seq}[U]) : \text{RDD}[T] \Rightarrow \text{RDD}[U]$ $\text{sample}(\text{fraction} : \text{Float}) : \text{RDD}[T] \Rightarrow \text{RDD}[T]$ (Deterministic sampling) $\text{groupByKey}() : \text{RDD}[(K, V)] \Rightarrow \text{RDD}[(K, \text{Seq}[V])]$ $\text{reduceByKey}(f : (V, V) \Rightarrow V) : \text{RDD}[(K, V)] \Rightarrow \text{RDD}[(K, V)]$ $\text{union}() : (\text{RDD}[T], \text{RDD}[T]) \Rightarrow \text{RDD}[T]$ $\text{join}() : (\text{RDD}[(K, V)], \text{RDD}[(K, W)]) \Rightarrow \text{RDD}[(K, (V, W))]$ $\text{cogroup}() : (\text{RDD}[(K, V)], \text{RDD}[(K, W)]) \Rightarrow \text{RDD}[(K, (\text{Seq}[V], \text{Seq}[W]))]$ $\text{crossProduct}() : (\text{RDD}[T], \text{RDD}[U]) \Rightarrow \text{RDD}[(T, U)]$ $\text{mapValues}(f : V \Rightarrow W) : \text{RDD}[(K, V)] \Rightarrow \text{RDD}[(K, W)]$ (Preserves partitioning) $\text{sort}(c : \text{Comparator}[K]) : \text{RDD}[(K, V)] \Rightarrow \text{RDD}[(K, V)]$ $\text{partitionBy}(p : \text{Partitioner}[K]) : \text{RDD}[(K, V)] \Rightarrow \text{RDD}[(K, V)]$
Actions	$\text{count}() : \text{RDD}[T] \Rightarrow \text{Long}$ $\text{collect}() : \text{RDD}[T] \Rightarrow \text{Seq}[T]$ $\text{reduce}(f : (T, T) \Rightarrow T) : \text{RDD}[T] \Rightarrow T$ $\text{lookup}(k : K) : \text{RDD}[(K, V)] \Rightarrow \text{Seq}[V]$ (On hash/range partitioned RDDs) $\text{save}(\text{path} : \text{String}) : \text{Outputs RDD to a storage system, e.g., HDFS}$

Table 2: Transformations and actions available on RDDs in Spark. Seq[T] denotes a sequence of elements of type T.

Word Count Example

```
counts = lines.flatMap(Lambda x: x.split(' '))  
                .map(Lambda x: (x, 1))  
                .reduceByKey(add)
```



Collaborative Filtering (CF)

- **The most prominent approach to generate recommendations**
 - used by large, commercial e-commerce sites
 - well-understood, various algorithms and variations exist
 - applicable in many domains (book, movies, DVDs, ..)
- **Approach**
 - use the "wisdom of the crowd" to recommend items
- **Basic assumption and idea**
 - Users give ratings to catalog items (implicitly or explicitly)
 - Customers who had similar tastes in the past, will have similar tastes in the future



User-based nearest-neighbor collaborative filtering (1)

- **The basic technique:**

- Given an "active user" (Alice) and an item I not yet seen by Alice
- The *goal is to estimate Alice's rating for this item*, e.g., by
 - find a set of users (peers) who liked the same items as Alice in the past **and** who have rated item I
 - use, e.g. the average of their ratings to predict, if Alice will like item I
 - do this for all items Alice has not seen and recommend the best-rated

	Item1	Item2	Item3	Item4	Item5
Alice	5	3	4	4	?
User1	3	1	2	3	3
User2	4	3	4	3	5
User3	3	3	1	5	4
User4	1	5	5	2	1

Measuring user similarity

- A popular similarity measure in user-based CF:

$$Sim(A, B) = \frac{A \cdot B}{||A|| \cdot ||B||}$$

$$||A|| = \sqrt{A \cdot A}$$


a, b : users

$r_{a,p}$: rating of user a for item p

P : set of items, rated both by a and b

Possible similarity values between -1 and 1; $\overline{r_a}, \overline{r_b}$ = user's average ratings

	Item1	Item2	Item3	Item4	Item5
Alice	5	3	4	4	?
User1	3	1	2	3	3
User2	4	3	4	3	5
User3	3	3	1	5	4
User4	1	5	5	2	1



sim = 0,85
sim = 0,70
sim = -0,79

Very easy approach, but...

Memory-based Approach

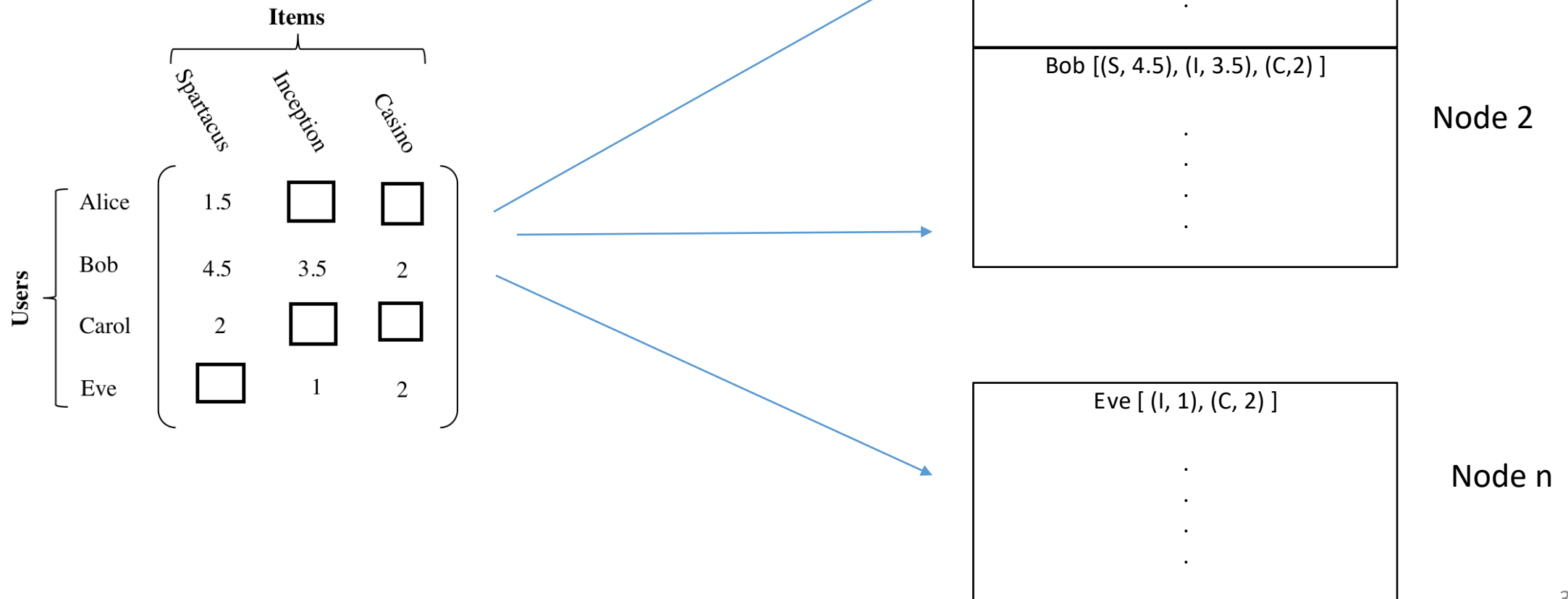
- **User-based CF is said to be "memory-based"**
 - the rating matrix is directly used to find neighbors / make predictions
 - does not scale for most real-world scenarios
 - large e-commerce sites have tens of millions of customers and millions of items

Movielens :

1. **10 million** ratings applied to 10,000 movies by 72,000 users
2. The size of rating matrix: $10\text{ k} * 72\text{ k} = \mathbf{720\text{ M}}$

memory-based model in distributed platform

- Sparse matrix



Similarity metric

loadFile.map().groupByKey()

USER	MOVIE	Rating
11	100	5
11	101	3
11	102	4
12	101	2
12	102	4
12	103	1
13	100	5
13	101	3
13	103	4

100	[(11, 5), (13, 5) ...]
101	[(11, 3), (12, 2), (13, 3) ...]
102	[(11, 4), (12, 4) ...]
103	[(12, 1), (13, 4) ...]

map().groupByKey()

100 [(11, 5), (13, 5) - -]
101 [(11, 3), (12, 2) (13, 3) -]
102 [(11, 4), (12, 4) - -]
103 [(12, 1), (13, 4) - -]

(11, 13) [(5, 5), (3, 3) -]
(11, 12) [(3, 2), (4, 4) - -]
(12, 13) [(2, 3), (1, 4) - -]

`map(calcSim()).map().groupByKey()`

$(11, 13) \quad [(5, 5), (3, 3)]$

$(11, 12) \quad [(3, 2), (4, 4)]$

$(12, 13) \quad [(2, 3), (1, 4)]$

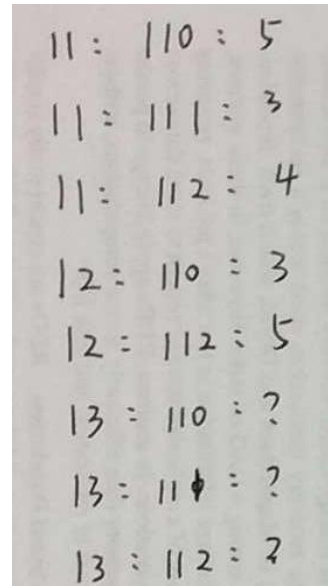
$11 \quad [(12, \text{sim}), (13, \text{sim})]$

$12 \quad [(11, \text{sim}), (13, \text{sim})]$

$13 \quad [(11, \text{sim}), (12, \text{sim})]$

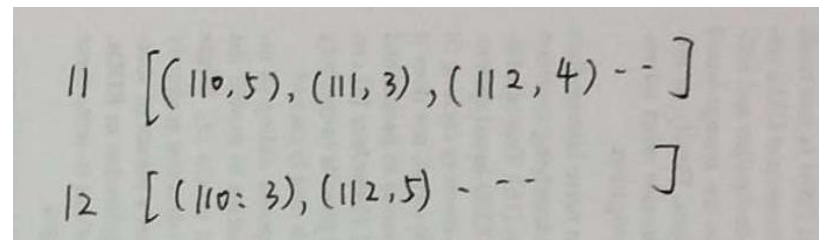
Load history record

`loadFile.map().groupByKey()`



Handwritten load history records showing key-value pairs for keys 11, 12, and 13. The records are as follows:

Key	Value	Count
11	110	5
11	111	3
11	112	4
12	110	3
12	112	5
13	110	?
13	111	?
13	112	?



Handwritten grouped load history records showing the data organized by key into lists of (key, value, count) tuples:

Key	Records
11	$[(110, 5), (111, 3), (112, 4) \dots]$
12	$[(110, 3), (112, 5) \dots]$

Neighbor recommendation

```
User [(movie,rating),(movie,rating)]  
122 [(1,1),(5,1),(22,-1).....]  
185 [(1,1),(7,1),(52,-1).....]
```

.
. .
. .
. .
. .

Historical data
(broadcast)

```
User1 [(user2,similarity2),(user3,similarity3).....]
```

.
. .
. .
. .
. .

Similarity model