# Chapter 5. Building a Recommendation Engine with Spark

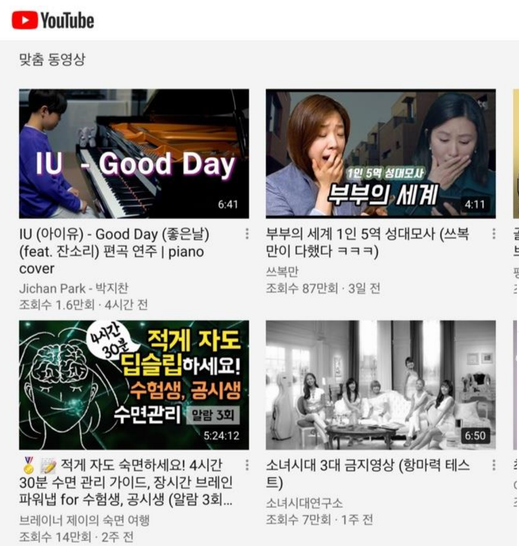
빅데이터분석

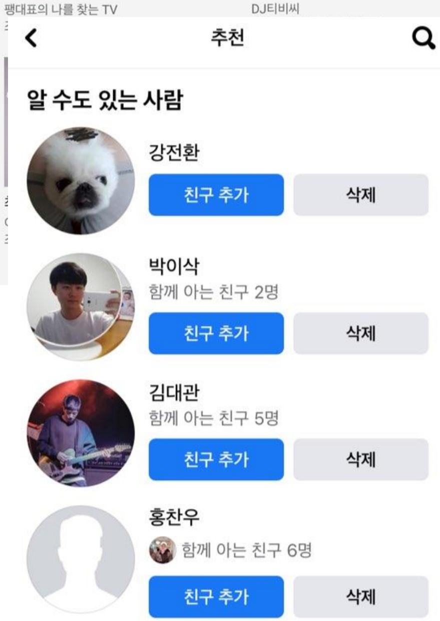
발표일 : 2020.04.22

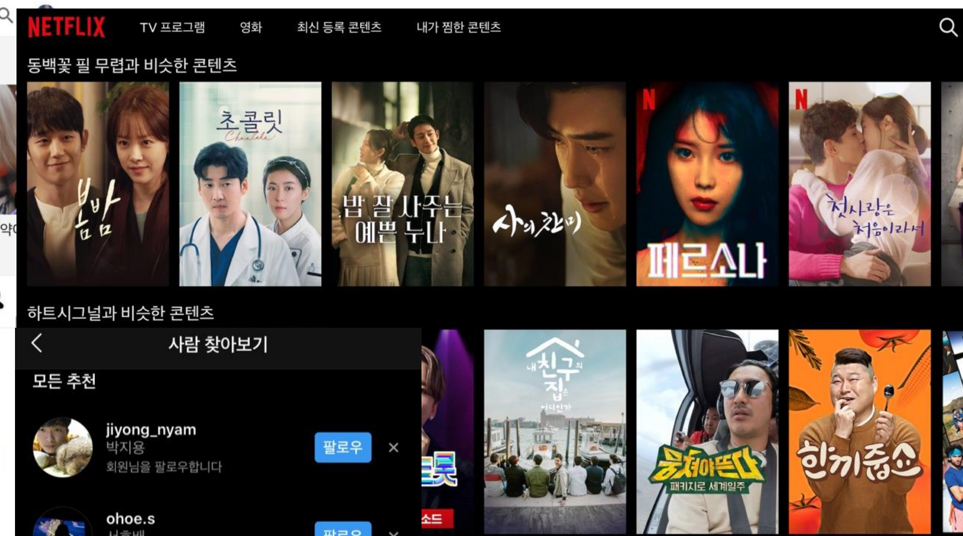
발표자 : 조소영

# Contents

# 1 Intro to Recommendation System



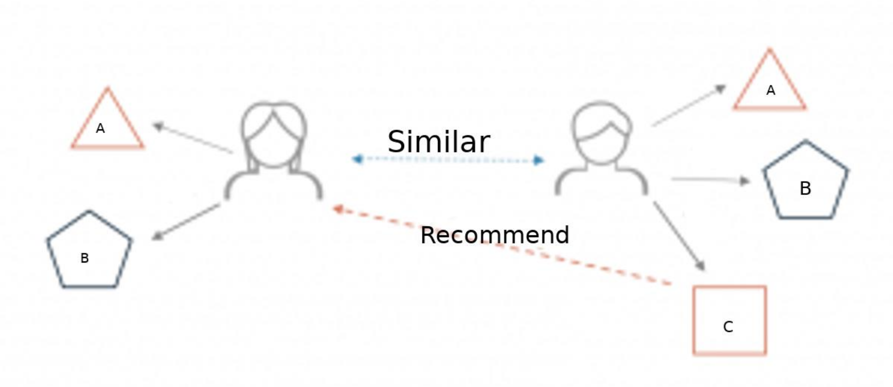

# 1 Intro to Recommendation System


Recommendations are a core part of all the businesses

# 1 Intro to Recommendation System

- **Basic Idea of Recommendation Engines**

- To predict what people might like
- To uncover relationships between items to aid in the discovery process
- Tries to model the connections between users and some type of item




- **Recommendation Engine VS Search Engines**

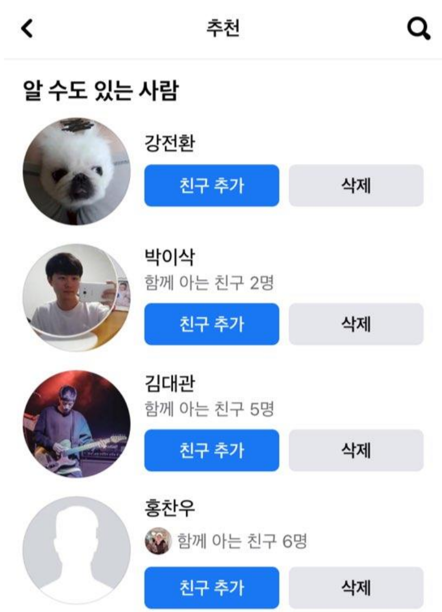
- Both plays a role in discovery
- Unlike Search engines, Recommendation engines try to present people with relevant content that they did not necessarily search for or that they might have not even heard of

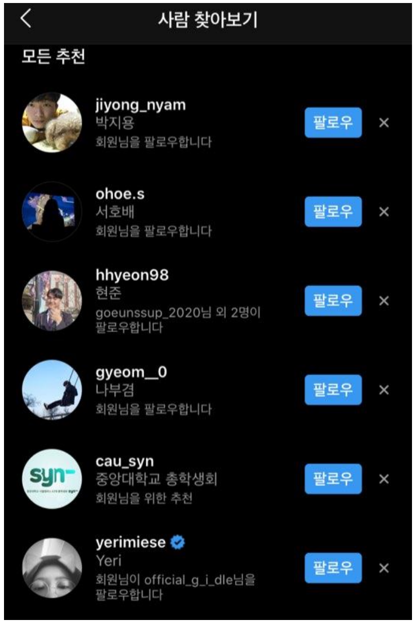
# 1 Intro to Recommendation System

- **Recommendation engines are not limited to items**

- Can be applied to just about any user-to-item relationship as well as user-to-user connections
- E.g) Social Networks

facebook




Instagram

# 1 Intro to Recommendation System

- **When are Recommendation Engines effective?**

Most effective in two general scenarios :

**1. Large number of available options for users**

- Becomes increasingly difficult to find something user wants
- Searching can help when user knows what they are looking for,

  But right item might be something previously unknown to them
- In this case, being recommended relevant items can help discover new items

# 1 Intro to Recommendation System

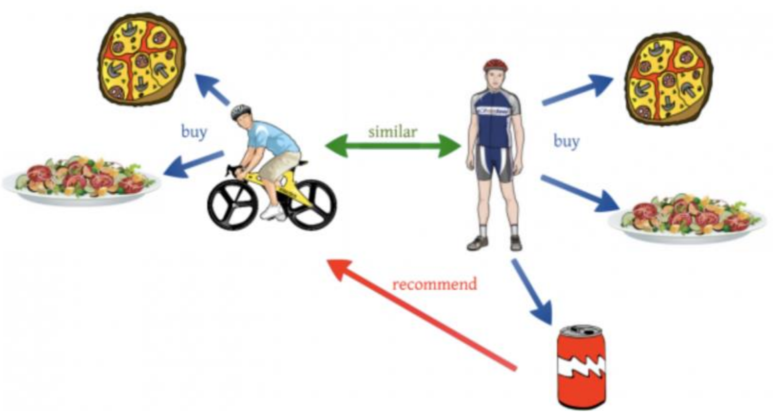
- **When are Recommendation Engines effective?**

Most effective in two general scenarios :

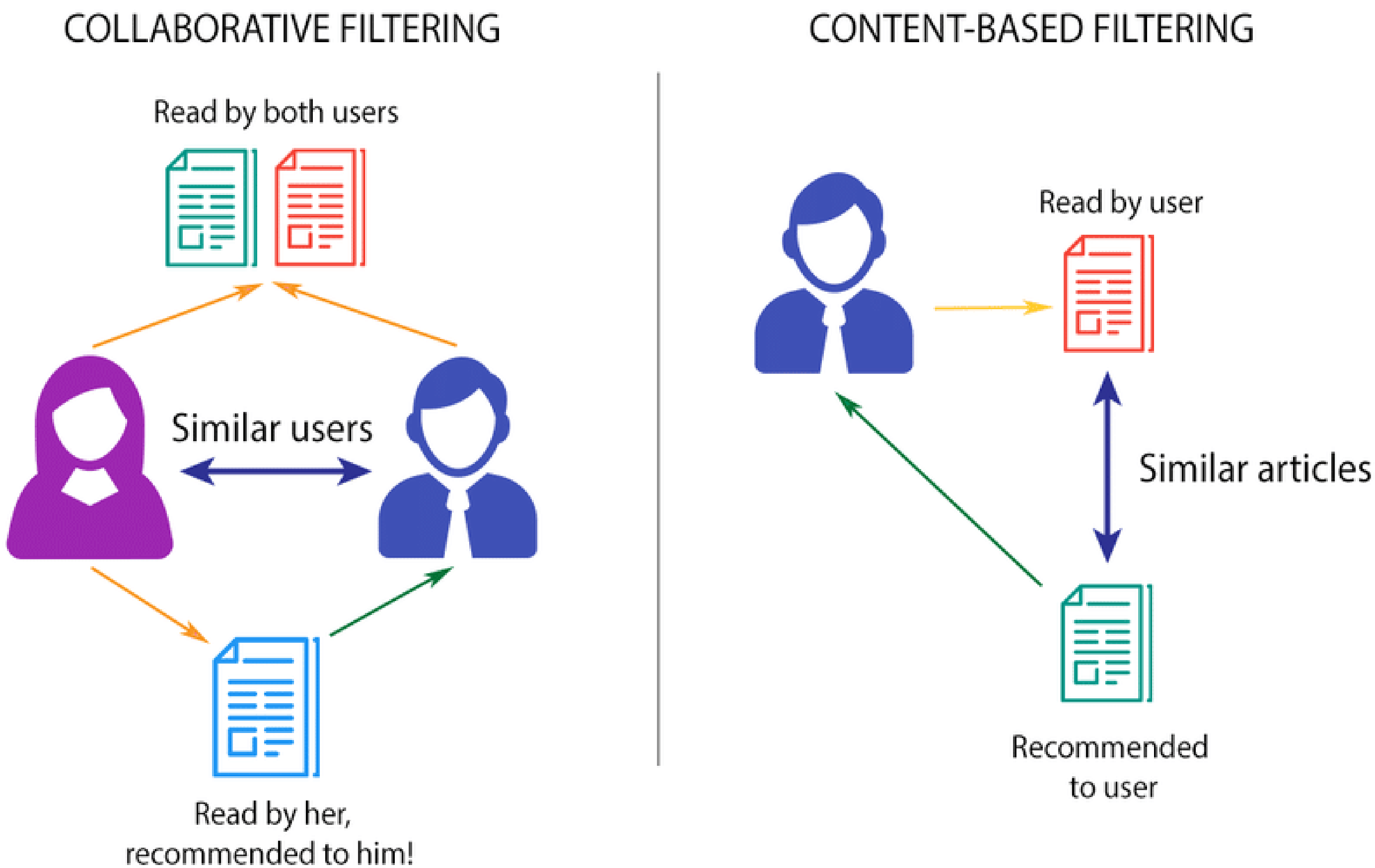
1. Large number of available options for users

**2. Significant degree of personal taste involved**

- Recommendation models can be helpful in discovering items based on the behavior of others that have similar taste profiles
- Often utilize a wisdom-of-the-crowd approach



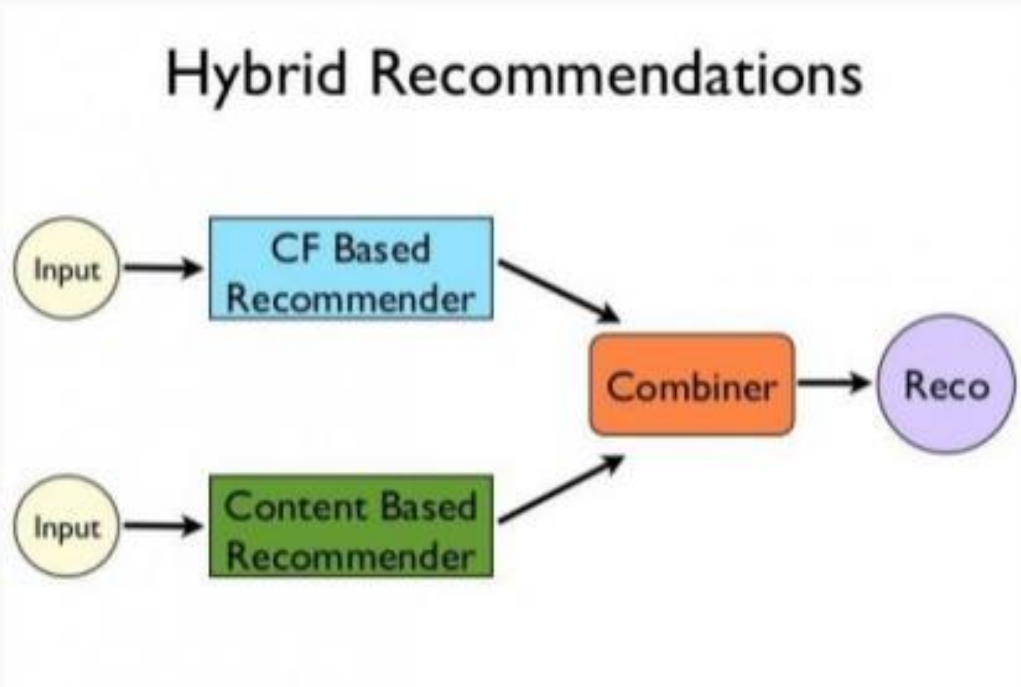
# 2 Types of recommendation models




- Recently other approaches have gained in popularity (ranking models)

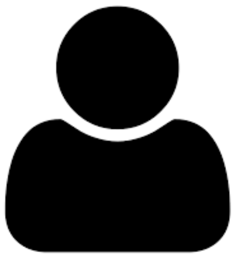
# 2 Types of recommendation models

In practice, many approaches are hybrids



# 2 Types of recommendation models

**Misunderstanding on**
**Content-based filtering VS Collaborative filtering**



Collaborative filtering

Content-based filtering

# 2 Types of recommendation models

## 1. Content-based filtering

### (1) In case of Item recommendations

Content-based methods try to use the content or attributes of an item, together with some notion of similarity between two pieces of content to generate items similar to a given item

### (2) In case of User recommendations

User recommendations can be generated based on attributes of users or user profiles, which are then matched to item attributes using the same measure of similarity

# 2 Types of recommendation models

## 1. Content-based filtering

Limits of Content-based filtering

| Movie | Alice (1) | Bob (2) | Carol (3) | Dave (4) | $x_1$ (romance) | $x_2$ (action) |
|---|---|---|---|---|---|---|
| Love at last | 5 | 5 | 0 | 0 | 0.9 | 0 |
| Romance forever | 5 | ? | ? | 0 | 1.0 | 0.01 |
| Cute puppies of love | ? | 4 | 0 | ? | 0.99 | 0 |
| Nonstop car chases | 0 | 0 | 5 | 4 | 0.1 | 1.0 |
| Swords vs. karate | 0 | 0 | 5 | ? | 0 | 0.9 |

# 2 Types of recommendation models

## 2. Collaborative filtering

- relies only on past behavior, such as previous ratings or transactions
- Notion of similarity

- User gives ratings to items, implicitly or explicitly
- Users who had a similar taste in the past will have a similar taste in the future

# 2 Types of recommendation models

## 2. Collaborative filtering

### Advantages of Collaborative filtering

- Easy to have dataset (compared to content-based filtering)
- Known to be more accurate than content-based filtering

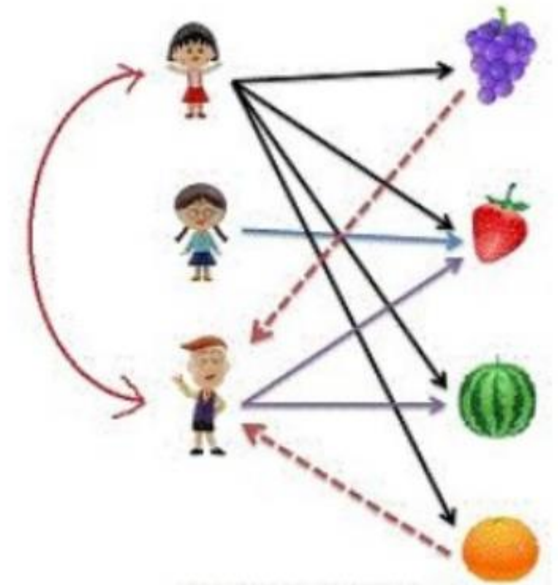
### Limits of Content-based filtering

- Less or none data for new user
- Cold start problem

# 2 Types of recommendation models

## 2. Collaborative filtering

### (1) User-based approach

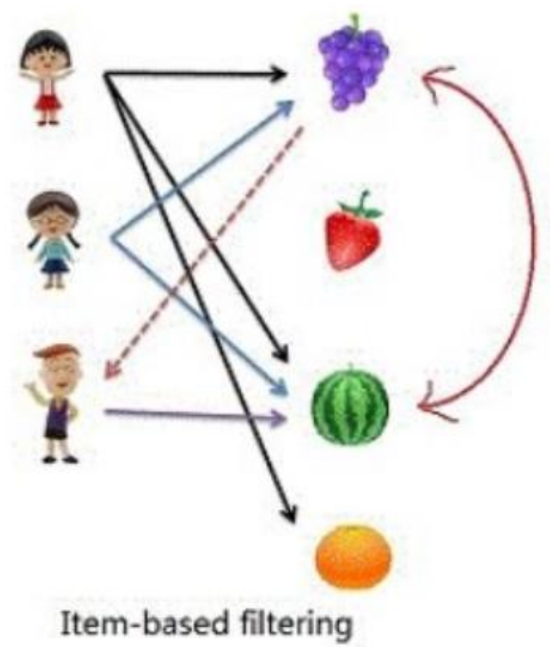



- If two users have shown similar preferences, We would assume they are similar to each other in terms of taste
- If others have tastes similar to a set of items, These items would tend to be good candidates for recommendation

# 2 Types of recommendation models

## 2. Collaborative filtering

### (2) Item-based approach




- Computes some measure of similarity between items
- Usually based on the existing user-item preferences or ratings
- Items that tend to be rated the same by similar users will be classed similar under this approach

# 2 Types of recommendation models

## 2. Collaborative filtering

### (3) Model-based approach

- attempt to model the user-item preferences themselves
- New preferences can be estimated directly by applying the model to unknown user-item combination

# 2 Types of recommendation models

## 2. Collaborative filtering

### (3) Model-based approach

1. **Neigborhood methods**

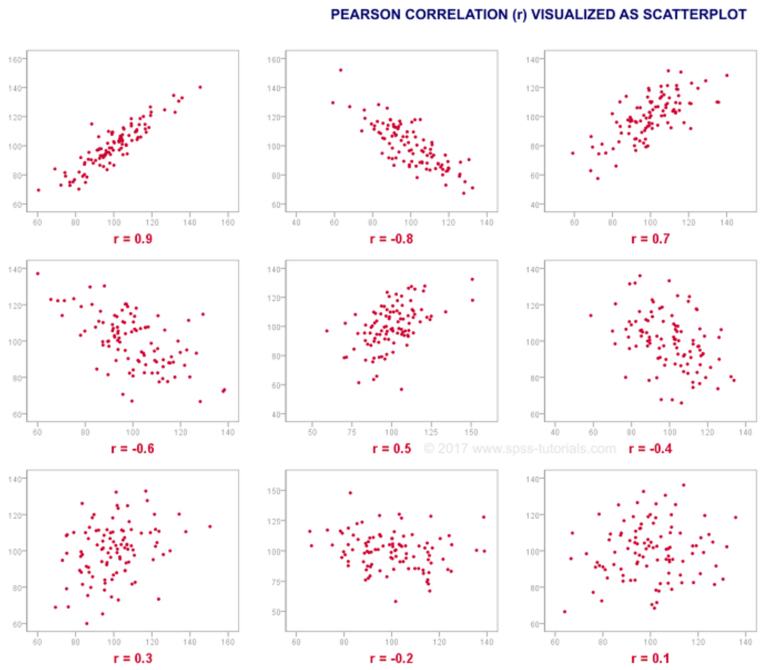
a. user-oriented approach

b. Item-oriented approach

- Use centered cosine distance for similarity calculation

(pearson correlation coefficients)

$$r_{XY} = \frac{\sum_{i=1}^{n}(X_i - \overline{X})(Y_i - \overline{Y})}{\sqrt{\sum_{i=1}^{n}(X_i - \overline{X})^2}\sqrt{\sum_{i=1}^{n}(Y_i - \overline{Y})^2}}$$



# 2 Types of recommendation models

## 2. Collaborative filtering

### (3) Model-based approach

**2. Latent factor models (LFM)**

- explains ratings by characterizing both users and items to find the hidden latent features
- e.g  Movies
- In Movies, features such as action or drama, type of actors are latent factors
- In Users, features such as liking the score for the movie is an example a latent factor
- Neural networks, latent Dirichlet allocation, **matrix factorization**

# 3 Matrix factorization

- Spark’s recommendation models currently only include an implementation of Matrix factorization
- These types of models have been shown to perform extremely well in collaborative filtering

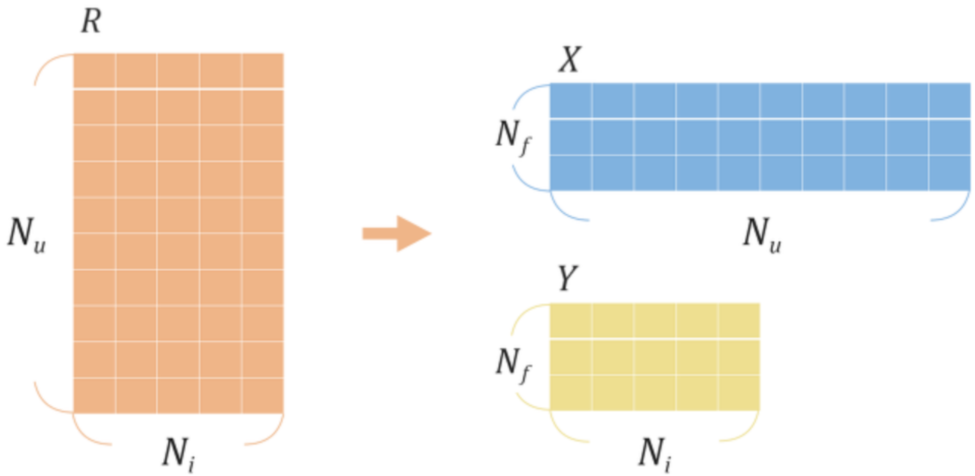
# 3 Matrix factorization

- **Assumption**

- Each user can be described by n attributes or features
- Each item can be described by a set of n attributes or features
- If we multiply each feature of the user by the corresponding feature of the item and add everything together, this will be a good approximation for the rating the user would give that item

# 3 Matrix factorization

## Basic Concept




$R$ : *original rating data matrix*

$N_u$ : *number of users*

$N_i$ : *number of items*

$N_f$ : *dimension of latent factor*

$X$ : *user lantent factor matrix* $(N_f \times N_u)$

$Y$ : *item latent factor matrix* $(N_f \times N_i)$

$x_u$ : *user latent vector of specific index* $u$

$y_i$ : *item latent vector of specific index* $i$

# 3 Matrix factorization

## Basic Concept

Item

| User | W | X | Y | Z |
|---|---|---|---|---|
| A | | 4.5 | 2.0 | |
| B | 4.0 | | 3.5 | |
| C | | 5.0 | | 2.0 |
| D | | 3.5 | 4.0 | 1.0 |

Rating Matrix

=

| | | |
|---|---|---|
| A | *1.2* | *0.8* |
| B | *1.4* | *0.9* |
| C | *1.5* | *1.0* |
| D | *1.2* | *0.8* |

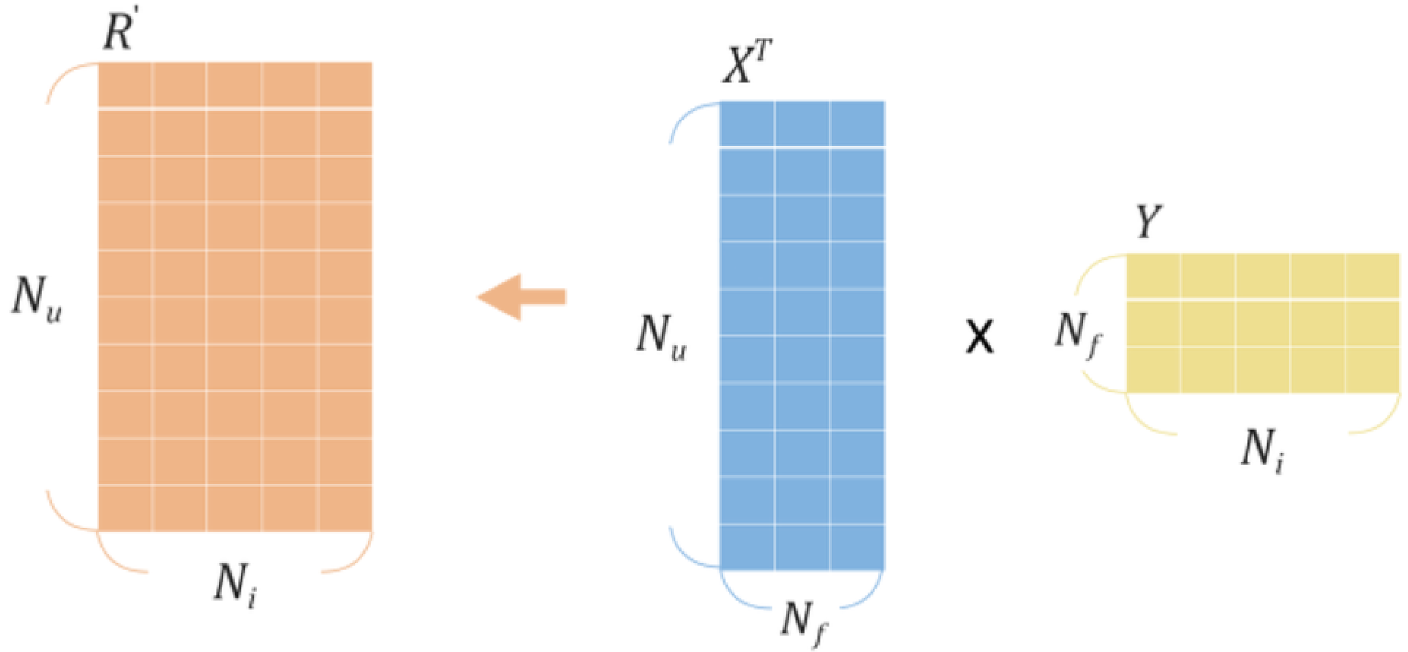
User Matrix

X

| W | X | Y | Z |
|---|---|---|---|
| *1.5* | *1.2* | *1.0* | *0.8* |
| *1.7* | *0.6* | *1.1* | *0.4* |

Item Matrix

# 3 Matrix factorization




$R'$ : *predicted rating data matrix*

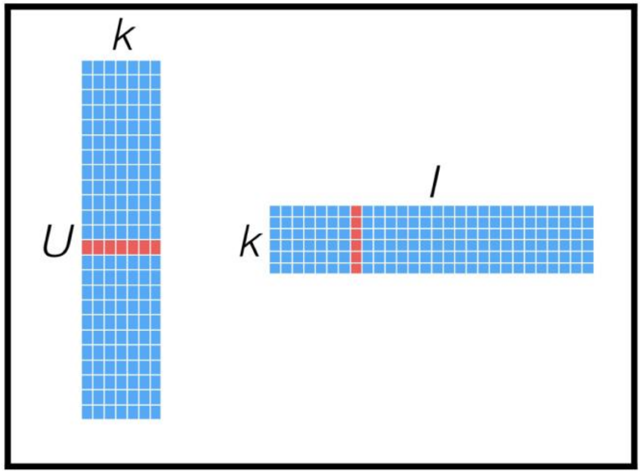
$$r_{ui} = x_u^T y_i$$

- Dot product between user latent vector $xu$ and item latent vector $yi$ captures the interaction between user and item == user's interest in an item
- Goal is to find vectors

# 3 Matrix factorization

## Loss Function

$$\min_{x^*,\ y^*} \sum_{u,i} (r_{ui} - x_u^T y_i)^2 + \lambda$$



# 3 Matrix factorization

## Optimization

- Learning algorithms used are stochastic gradient descent (SGD) or Alternating Least Squares (ALS)

### 1. stochastic gradient descent (SGD)

**Optimization algorithm:**

$$\min_{\theta^{(1)},\ldots,\theta^{(n_u)}} \frac{1}{2}\sum_{j=1}^{n_u}\sum_{i:r(i,j)=1}\left((\theta^{(j)})^T x^{(i)} - y^{(i,j)}\right)^2 + \frac{\lambda}{2}\sum_{j=1}^{n_u}\sum_{k=1}^{n}(\theta_k^{(j)})^2$$

Gradient descent update:

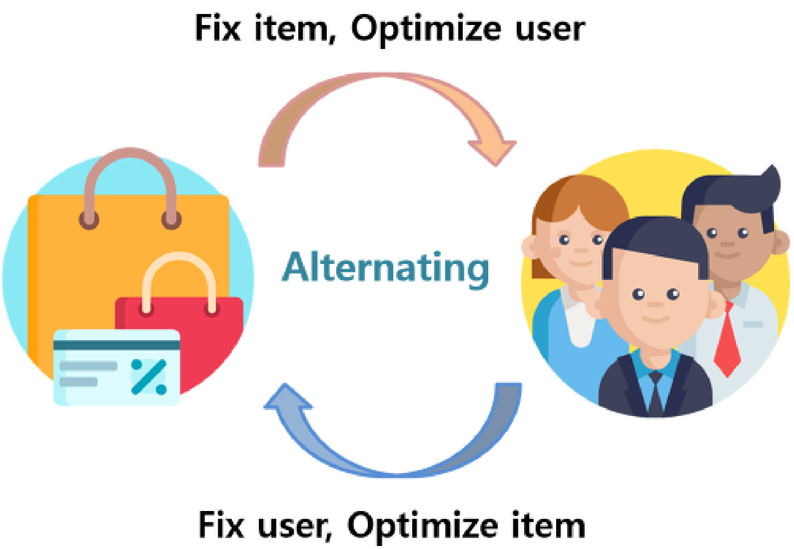
$$\theta_k^{(j)} := \theta_k^{(j)} - \alpha \sum_{i:r(i,j)=1}((\theta^{(j)})^T x^{(i)} - y^{(i,j)})x_k^{(i)} \text{ (for } k = 0)$$

$$\theta_k^{(j)} := \theta_k^{(j)} - \alpha \left(\sum_{i:r(i,j)=1}((\theta^{(j)})^T x^{(i)} - y^{(i,j)})x_k^{(i)} + \lambda\theta_k^{(j)}\right) \text{ (for } k \neq 0)$$

But, Objective function is not convex since both x and y are not known.

# 3 Matrix factorization

## 2. ALS (Alternating Least Squares)




- ALS is implemented in Apache Spark ML and built for a larges-scale collaborative filtering problems
- ALS is doing a pretty good job at solving scalability and sparseness of the Ratings data, and it’s simple and scales well to very large datasets.

# 3 Matrix factorization

Some high-level ideas behind ALS are:

•Its training routine is different: ALS minimizes **two loss functions alternatively**; It first holds user matrix fixed and runs gradient descent with item matrix; then it holds item matrix fixed and runs gradient descent with user matrix

• Its scalability: ALS runs its gradient descent in **parallel** across multiple partitions of the underlying training data from a cluster of machines

# 3 Matrix factorization

## Steps of ALS implementation

Step 1. In each iteration, one of the user- or item-factor matrices is treated fixed, while the other one is updated using the fixed factor and the rating data

Step 2. The factor matrix that was solved for is treated as fixed,

While the other one is updated using the fixed factor and the rating data.

Step 3. Continue until the model has converged / for a fixed num. of iterations

# 4 Extracting features

**From the MovieLens 100k dataset**

- use explicit rating data, without additional user or item metadata or other information related to the user-item interactions
- Hence, the features that we need as inputs are simply the

user IDs, movie IDs, and the ratings assigned to each user and movie pair.

# 4 Extracting features

## From the MovieLens 100k dataset

```
scala> val rawData =
sc.textFile("hdfs:///user/cloudera/movielens/u.data")
scala> rawData.first()
```

```
res0: String = 196	242	3	881250949
```

```
scala> val rawRatings = rawData.map(_.split("\t").take(3))
scala> rawRatings.first()
```

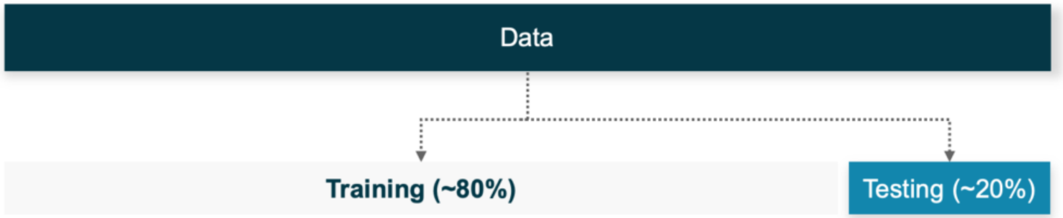
```
res2: Array[String] = Array(196, 242, 3)
```

```
scala> import org.apache.spark.mllib.recommendation.Rating
scala> val ratings = rawRatings.map { case Array(user, movie,
rating) =>Rating(user.toInt, movie.toInt, rating.toDouble) }
scala> ratings.first()
```

# 5 Training the recommendation model

## on the MovieLens 100k dataset

- we are ready to proceed with model training once we have extracted these simple features from our raw data,
- All we have to do is provide the correctly-parsed input RDD we just created as well as our chosen model parameters.
- Split dataset into training & test sets with 80:20



# 5 Training the recommendation model

**on the MovieLens 100k dataset**

- **numBlocks** is the number of blocks used to parallelize computation (set to -1 to autoconfigure)
- **rank** is the number of latent factors in the model
- **iterations** is the number of iterations to run
- **lambda** specifies the regularization parameter in ALS
- **implicitPrefs** specifies whether to use the explicit feedback ALS variant or one adapted for an implicit feedback data

# 5 Training the recommendation model

**on the MovieLens 100k dataset**

```
scala> import org.apache.spark.mllib.recommendation.ALS
scala> val model = ALS.train(ratings, 50, 10, 0.01)
```

Note: We'll use rank of 50, 10 iterations, and a lambda parameter of 0.01

- This returns a *MatrixFactorizationModel* object (ALSModel object)

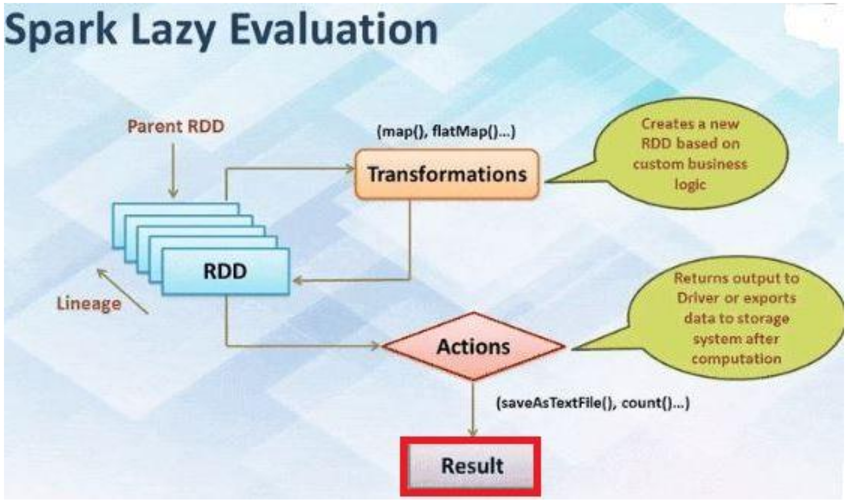
Which contains the user and item factors in the form of an RDD of *(id, factor)* pairs.

(*userFeatures , productFeatures)*

# 5 Training the recommendation model

## on the MovieLens 100k dataset

- Note that the operations used in MLlib's *ALS* implementation are lazy transformations, so the actual computation will only be performed once we call some sort of action on the resulting *RDDs* of the user and item factors.
- We can force the computation using a Spark action such as *count*:




- we have a factor array for each user (*943* factors) and movie (*1682* factors).

# 6 Using the recommendation model

- Now that we have our trained model, we're ready to use it to make predictions.
- Two forms of Prediction

(1) recommendations for a given user (user-item recommend)

(2) related or similar items for a given item (item-item recommend)

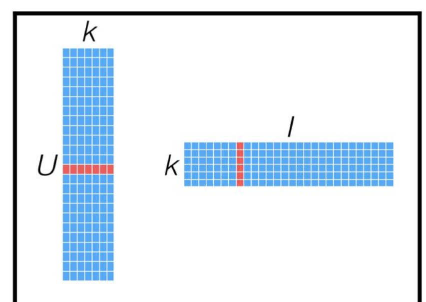
# 6 Using the recommendation model

## 1. User recommendations

- usually takes the form of a *top-K* list
- the *K* items that our model predicts will have the highest probability of the user liking them
- done by computing the predicted score for each item and ranking the list based on this score.

- **Computing the predicted score in matrix factorization**
- modeling the ratings matrix directly
- predicted score can be computed as the vector dot product between a user-factor vector and an item-factor vector

$$\min_{x^*,\, y^*} \sum_{u,i} (r_{ui} - \boxed{x_u^T y_i})^2 + \lambda$$



# 6 Using the recommendation model

## 1. User recommendations

### Generating movie recommendations from the MovieLens 100k dataset

- MLlib's recommendation model is based on matrix factorization
- can use the factor matrices computed by our model to compute predicted scores (or ratings) for a user

```
val predictedRating = model.predict(789, 123)

INFO SparkContext: Starting job: lookup at
MatrixFactorizationModel.scala:
INFO DAGScheduler: Got job 30 (lookup at
MatrixFactorizationModel.scala:45) with 1 output partitions
(allowLocal=false)
...

INFO SparkContext: Job finished: lookup at
MatrixFactorizationModel.scala:46, took 0.023077 s

predictedRating: Double = 3.128545693368485
```

# 6 Using the recommendation model

## 1. User recommendations

- To generate the *top-K* recommended items for a user, *MatrixFactorizationModel* provides a convenience method called *recommendProducts*.
- This takes two arguments: *user* and *num*, where *user* is the user ID, and *num* is the number of items to recommend.

```
val userId = 789
val K = 10
val topKRecs = model.recommendProducts(userId, K)

println(topKRecs.mkString("n"))

Rating(789,715,5.931851273771102)
Rating(789,12,5.582301095666215)
Rating(789,959,5.516272981542168)

…
```

# 6 Using the recommendation model

## 1. User recommendations

```
scala> val moviesForUser = ratings.keyBy(_.user).lookup(789)
```

```
moviesForUser: Seq[org.apache.spark.mllib.recommendation.Rating] = WrappedArray(Rati
ng(789,1012,4.0), Rating(789,127,5.0), Rating(789,475,5.0), Rating(789,93,4.0), Rati
ng(789,1161,3.0), Rating(789,286,1.0), Rating(789,293,4.0), Rating(789,9,5.0), Ratin
g(789,50,5.0), Rating(789,294,3.0), Rating(789,181,4.0), Rating(789,1,3.0), Rating(7
89,1008,4.0), Rating(789,508,4.0), Rating(789,284,3.0), Rating(789,1017,3.0), Rating
(789,137,2.0), Rating(789,111,3.0), Rating(789,742,3.0), Rating(789,248,3.0), Rating
(789,249,3.0), Rating(789,1007,4.0), Rating(789,591,3.0), Rating(789,150,5.0), Ratin
g(789,276,5.0), Rating(789,151,2.0), Rating(789,129,5.0), Rating(789,100,5.0), Ratin
g(789,741,5.0), Rating(789,288,3.0), Rating(789,762,3.0), Rating(789,628,3.0), Ratin
g(789,124,4.0))
```

For our user *789*, we can find out what movies they have rated, take the *10* movies with the highest rating, and then check the titles.

***keyBy*** : create an RDD of key-value pairs from our *ratings* RDD, where the key will be the user ID.

***lookup*** : function to return just the ratings for this key (particular user ID) to the driver

# 6 Using the recommendation model

## 1. User recommendations

Next, we will take the 10 movies with the highest ratings by sorting the *moviesForUser* collection using the *rating* field of the *Rating* object.

Extract the movie title for the relevant product ID attached to the *Rating* class from our mapping of movie titles and print out the top *10* titles with their ratings

```
scala> moviesForUser.sortBy(-_.rating).take(10).map(rating =>
(titles(rating.product), rating.rating)).foreach(println)
```

```
(Godfather, The (1972),5.0)
(Trainspotting (1996),5.0)
(Dead Man Walking (1995),5.0)
(Star Wars (1977),5.0)
(Swingers (1996),5.0)
(Leaving Las Vegas (1995),5.0)
(Bound (1996),5.0)
(Fargo (1996),5.0)
(Last Supper, The (1995),5.0)
(Private Parts (1997),4.0)
```

# 6 Using the recommendation model

## 1. User recommendations

Take a look at what the titles are using the same approach as the one we used earlier (note that the recommendations are already sorted)

```
scala> val topKRecs = model.recommendProducts(789,10)

scala> topKRecs.map(rating => (titles(rating.product),
rating.rating)).foreach(println)
```

```
(GoodFellas (1990),5.561893309975536)
(Apocalypse Now (1979),5.359509740087787)
(Being There (1979),5.253109995320087)
(Carrie (1976),5.214960672591296)
(Aliens (1986),5.18467232737804)
(Psycho (1960),5.184123552034558)
(One Flew Over the Cuckoo's Nest (1975),5.174956083257432)
(Full Monty, The (1997),5.145369582639113)
(Flirting With Disaster (1996),5.128468420256269)
(Heavy Metal (1981),5.112027118820185)
```

# 6 Using the recommendation model

## 2. Item recommendations

- Is about answering the following question:

*for a certain item, what are the items most similar to it?*

- the precise definition of similarity is dependent on the model involved.

- In most cases, similarity is computed by comparing the vector representation of two items using some similarity measure.

- Common similarity measures include :

for real-valued vectors -> Pearson correlation and cosine similarity
for binary vectors -> Jaccard similarity.

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}$$

# 6 Using the recommendation model

## 2. Item recommendations

### Generating movie recommendations from the MovieLens 100k dataset

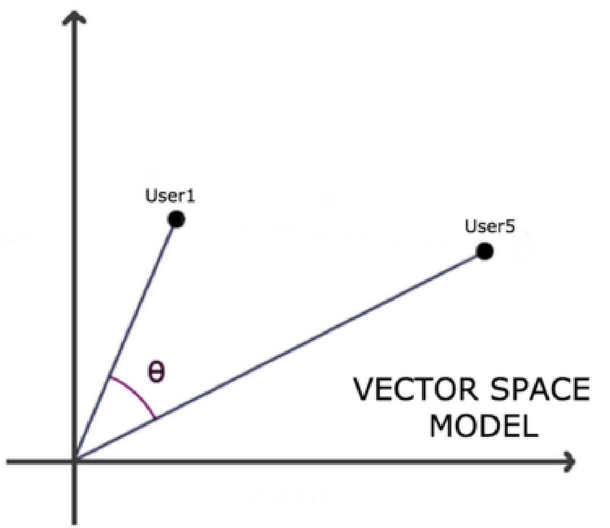
- The current *MatrixFactorizationModel* API does not directly support item-to-item similarity computations
- Therefore, we will need to create our own code to do this.

- use the cosine similarity metric
- use the jblas linear algebra library (a dependency of MLlib) to compute the required vector dot products.
- similar to how the existing *predict* and *recommendProducts* methods work, except that we will use cosine similarity as opposed to just the dot product.

# 6 Using the recommendation model

## 2. Item recommendations

### Generating movie recommendations from the MovieLens 100k dataset




$$s(\mathbf{x},\mathbf{y}) = \frac{\mathbf{x}\cdot\mathbf{y}}{\|\mathbf{x}\|\|\mathbf{y}\|} = \frac{\sum_{i=0}^{n-1} x_i y_i}{\sqrt{\sum_{i=0}^{n-1}(x_i)^2} \times \sqrt{\sum_{i=0}^{n-1}(y_i)^2}}$$

```
def cosineSimilarity(vec1: DoubleMatrix, vec2: DoubleMatrix): Double = {
    vec1.dot(vec2) / (vec1.norm2() * vec2.norm2())
}
```

# 6 Using the recommendation model

## 2. Item recommendations

```
val itemId = 567
val itemFactor = model.productFeatures.lookup(itemId).head
val itemVector = new DoubleMatrix(itemFactor)

val sims = model.productFeatures.map{ case (id, factor) =>

 val factorVector = new DoubleMatrix(factor)

  val sim = cosineSimilarity(factorVector, itemVector)

  (id, sim)

}

val sortedSims = sims.top(K)(Ordering.by[(Int, Double), Double] {
 case (id, similarity) => similarity })
```

# 6 Using the recommendation model

## 2. Item recommendations

```
val sortedSims2 = sims.top(K + 1)(Ordering.by[(Int, Double), Double]
{ case (id, similarity) => similarity })

sortedSims2.slice(1, 11).map{ case (id, sim) => (titles(id),
sim) }.mkString("n")
```

- As we did for user recommendations, we can sense check our item-to-item similarity computations and take a look at the titles of the most similar movies.
- we will take the top 11 so that we can exclude our given movie

# 6 Using the recommendation model

## 2. Item recommendations

```
println(titles(itemId))
Wes Craven's New Nightmare (1994)

=======

(Hideaway (1995),0.6932331537649621)
(Body Snatchers (1993),0.6898690594544726)
(Evil Dead II (1987),0.6897964975027041)
(Alien: Resurrection (1997),0.6891221044611473)
(Stephen King's The Langoliers (1995),0.6864214133620066)
(Liar Liar (1997),0.6812075443259535)
(Tales from the Crypt Presents: Bordello of Blood(1996),0.6754663844488256)
(Army of Darkness (1993),0.6702643811753909)
(Mystery Science Theater 3000: The Movie (1996),0.6594872765176396)
(Scream (1996),0.6538249646863378)
```

# Thank You