Building a Recommendation Engine with Spark

Machine Learning with Spark

Nick Pentreath

Hong Cheng
chenghong@kw.ac.kr
Advisor: Prof. Hyukjoon Lee

Computer Intelligence Network Laboratory, Dept. of CE, KwangWoon University

CINe Lab

Contents

- Types of recommendation models
- Extracting the right features from your data
- Training the recommendation model
- Using the recommendation model
- Evaluating the performance of recommendation models

Introduction to Recommendation Engine

- Best types of machine learning model known to general public

- Importance
  - Keep our users engaged using our service
  - Improving our users' experience, engagement, and the relevance of our content to them
- Idea
  - Predict what people might like
  - Uncover relationships between items to aid in the discovery process
- Different from search engines
  - Present people with relevant content that they did not necessarily search for or that they might not even have heard of

Introduction to Recommendation Engine

- Goal
  - Model the connections between users and items
- Dataset
  - MovieSream with MovieLens 100k dataset
    - *943* users and *1682* movies
- Application scenario
  - User-to-item relationship and user-to-user connections
  - Large number of available options for users
    - Discover new items
  - A significant degree of personal taste involved

Types of recommendation models

- **Two prevalent approaches**
  - **Content-based filtering**
  - **Collaborative filtering**
  - **Ranking model**

Types of recommendation models

- **Content-based filtering**
  - **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.**
  - **Attributes**
    - **textual content, such as titles, names, tags, and other metadata attached to an item**
    - **Media features of the item, such as attributes extracted from audio and video content**

Types of recommendation models

- **Collaborative filtering**
  - **Idea – notion of similarity**
    - **User-based:** The overall logic is that if others have tastes similar to a set of items, these items would tend to be good candidates for recommendation.
    - **Item-based**: Computes some measure of similarity between items.
      - Based on the existing user-item preferences or ratings
      - Items(rated the same by similar user) → Similar → Similarity → Represent a user in terms of items → Recommend these items for the similar user
      - Use similar items to generate a combined score to estimate for an unknown item

Types of recommendation models

- **Collaborative filtering**
  - **The user/item-based approaches are usually referred to as nearest-neighbor models**
    - **Since the estimated scores are computed based on the set of most similar users or items (that is, their neighbors).**
    - **One of Spark's recommendation models**
- **Matrix factorization**
  - **Explicit matrix factorization**
  - **Implicit matrix factorization**
  - **Alternating least squares (ALS)**

Collaborative filtering

- **Matrix factorization**
  - **The user/item-based approaches are usually referred to as nearest-neighbor models**
    - **since the estimated scores are computed based on the set of most similar users or items (that is, their neighbors).**
    - **One of Spark's recommendation models**
- **Explicit matrix factorization**
- **Implicit matrix factorization**
- ***Alternating least squares (ALS)***

MLlib: Main Guide

- Pipelines
- Extracting, transforming and selecting features
- Classification and Regression
- Clustering
- Collaborative filtering
- Model selection and tuning
- Advanced topics

MLlib: RDD-based API Guide

- Data types
- Basic statistics
- Classification and regression
- Collaborative filtering
- Clustering
- Dimensionality reduction
- Feature extraction and transformation
- Frequent pattern mining
- Evaluation metrics
- PMML model export
- Optimization (developer)

MLlib: RDD-based API

This page documents sections of the MLlib guide for the RDD-based API (the `spark.mllib` package). Please see the MLlib Main Guide for the DataFrame-based API (the `spark.ml` package), which is now the primary API for MLlib.

- Data types
- Basic statistics
  - summary statistics
  - correlations
  - stratified sampling
  - hypothesis testing
  - streaming significance testing
  - random data generation
- Classification and regression
  - linear models (SVMs, logistic regression, linear regression)
  - naive Bayes
  - decision trees
  - ensembles of trees (Random Forests and Gradient-Boosted Trees)
  - isotonic regression
- Collaborative filtering
  - alternating least squares (ALS)
- Clustering
  - k-means
  - Gaussian mixture
  - power iteration clustering (PIC)
  - latent Dirichlet allocation (LDA)
  - bisecting k-means
  - streaming k-means
- Dimensionality reduction
  - singular value decomposition (SVD)
  - principal component analysis (PCA)
- Feature extraction and transformation
- Frequent pattern mining
  - FP-growth
  - association rules
  - PrefixSpan
- Evaluation metrics
- PMML model export
- Optimization (developer)
  - stochastic gradient descent
  - limited-memory BFGS (L-BFGS)

Collaborative filtering

Explicit matrix factorization

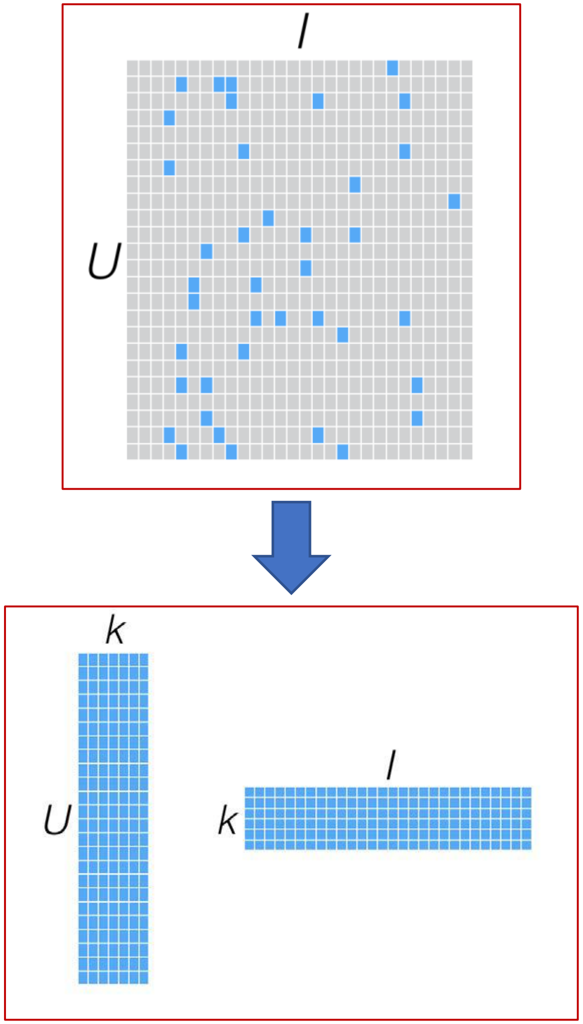
- **Definition**
  - The data with preference of users that provide by the user themselves
  - Rating, thumbs up, like.
- **Get Matrix**
  - Take the ratings
  - Form a 2-D matrix with users as rows and items as columns
- **The matrix is sparse**
  - Since in most cases, each user has only interacted with a relatively small set of items, this matrix has only a few non-zero entries

```
Tom, Star Wars, 5
Jane, Titanic, 4
Bill, Batman, 3
Jane, Star Wars, 2
Bill, Titanic, 3
```

User / Item	Batman	Star Wars	Titanic
Bill	3	3	
Jane		2	4
Tom		5	

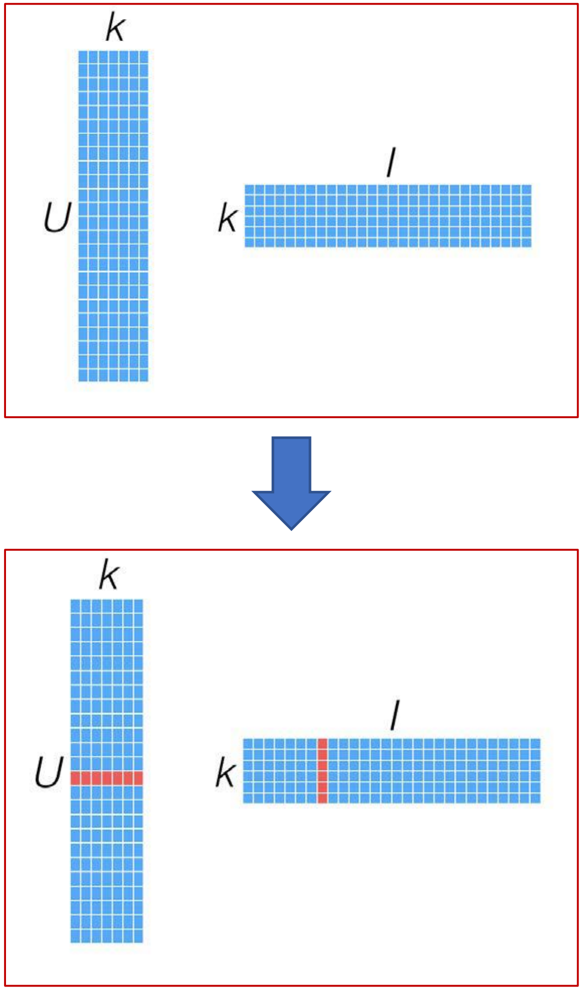
Collaborative filtering

- **Explicit matrix factorization**
  - **Definition**
    - Directly model user-item matrix by representing it as a product of two smaller matrices of low dimensions
    - It is a Dimensionality-Reduction technique.
  - **Example**
    - A user-item matrix (*U*I*)
    - Two factor matrices (*U*k, k*I*)
    - while the original ratings matrix is typically very sparse
    - Each factor matrix is dense



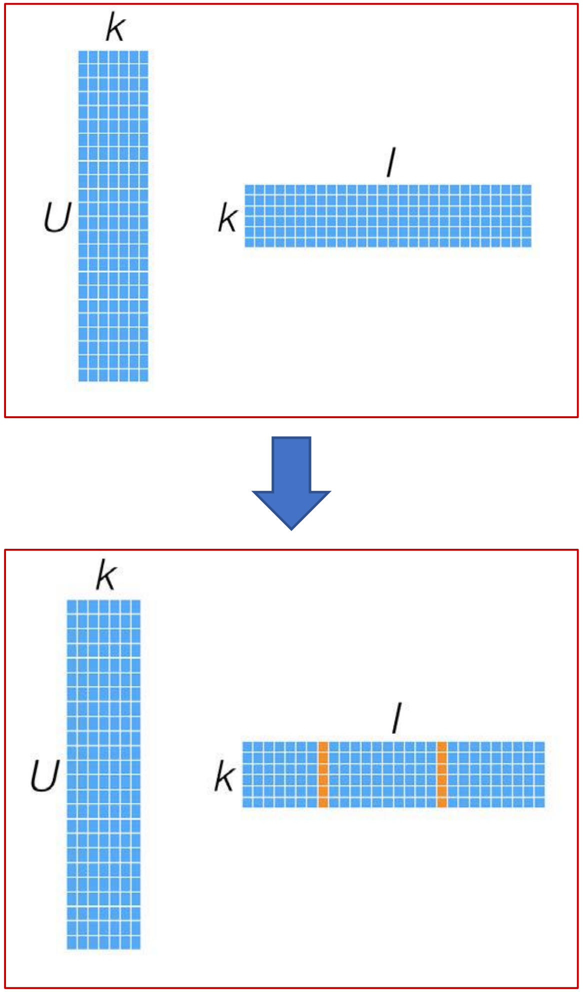
Collaborative filtering

- **Explicit matrix factorization**
  - **Latent feature models**
    - to discover **hidden features** (factor matrices) that account for the **structure of behavior inherent** in the user-item rating matrix.
  - **Compute prediction**
    - To compute a **predicted rating** for a user and item
    - Compute the **vector dot product** between the relevant row of the **user-factor matrix** and the relevant row of the **item-factor matrix**



Collaborative filtering

- **Explicit matrix factorization**
  - **Prediction in the model**
    - To find out the similarity between two items
    - Use the factor vectors directly by computing the similarity between two item-factor vectors
  - **Benefit**
    - The relative ease of computing recommendation once the model is created
    - Offer very good performance
  - **Challenge**
    - Require storage and computation across potentially many millions of user/item-factor vectors
    - more complex to understand and interpret
    - Computationally intensive during training



Collaborative filtering

- **Implicit matrix factorization**
  - **Implicit data**
    - between a user/item are not given to us
    - Implied from the interactions they might have with an item
    - Such as "whether/if" or count data
  - **How to deal with implicit data**
    - treats the input rating matrix as two matrices
      - a binary preference matrix, P,
      - a matrix of confidence weights, C.

```
Tom, Star Wars, 5
Jane, Titanic, 4
Bill, Batman, 3
Jane, Star Wars, 2
Bill, Titanic, 3
```

User / Item	Batman	Star Wars	Titanic
Bill	3	3	
Jane		2	4
Tom		5	

Collaborative filtering

- **Implicit matrix factorization**
  - **Assumption**
    - assume that the user-movie ratings were the number of times each user had viewed that movie.
    - Matrix P informs us that a movie was viewed by a user
    - Matrix C represents confidence weight
  - Generally, in the form of the view counts, the more a user has watched a movie, the higher the confidence that they actually like it.

P

User / Item	Batman	Star Wars	Titanic
Bill	1	1	
Jane		1	1
Tom		1	

C

User / Item	Batman	Star Wars	Titanic
Bill	3	3	
Jane		2	4
Tom		5	

Collaborative filtering

- Implicit matrix factorization
  - still creates a user/item-factor matrix
  - the model is attempting to approximate is
    - not the overall ratings matrix
    - but the preference matrix P
  - If we compute a recommendation
    - by calculating the dot product of a user/item-factor vector,
    - the score will not be an estimate of a rating directly.
    - It will rather be an estimate of the preference of a user for an item.

Collaborative filtering

- **Alternating least squares (ALS)**
  - An optimization technique to solve matrix factorization problems
    - Powerful
    - Achieves good performance
    - Proven to be relatively easy to implement in a parallel fashion(Spark@MLlib)
  - Works by iteratively solving least squares regression problems
    - In each iteration, one of the user/item-factor matrices is treated as fixed, while the other one is updated using the fixed factor and the rating data.
    - Then, the factor matrix that was solved for is, in turn, treated as fixed, while the other one is updated.
    - This process continues until the model has converged (or for a fixed number of iterations).
  - Spark ALS document
    - http://spark.apache.org/docs/latest/mllib-collaborative-filtering.html

Collaborative filtering

- **Alternating least squares (ALS)**
  - **spark.mllib uses ALS algorithm to learn these latent factors. The implementation has the following parameters:**
    - ***rank*** **is the number of latent factors in the model.**
    - ***iterations*** **is the number of iterations of ALS to run. ALS typically converges to a reasonable solution in 20 iterations or less.**
    - ***lambda*** **specifies the regularization parameter in ALS.**
    - ***numBlocks*** **is the number of blocks used to parallelize computation (set to -1 to auto-configure).**
    - ***implicitPrefs*** **specifies whether to use the explicit feedback ALS variant or one adapted for implicit feedback data.**
    - ***alpha*** **is a parameter applicable to the implicit feedback variant of ALS that governs the baseline confidence in preference observations.**

Collaborative filtering

Collaborative filtering

Given $x^{(1)}, \ldots, x^{(n_m)}$ (and movie ratings),
can estimate $\theta^{(1)}, \ldots, \theta^{(n_u)}$

Given $\theta^{(1)}, \ldots, \theta^{(n_u)}$,
can estimate $x^{(1)}, \ldots, x^{(n_m)}$

Guess $\Theta \rightarrow x \rightarrow \Theta \rightarrow x \rightarrow \Theta \rightarrow x \rightarrow \cdots$

Collaborative filtering optimization objective

$(i,j) : r(i,j)=1$

→ Given $x^{(1)},\dots,x^{(n_m)}$, estimate $\theta^{(1)},\dots,\theta^{(n_u)}$:

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

→ Given $\theta^{(1)},\dots,\theta^{(n_u)}$, estimate $x^{(1)},\dots,x^{(n_m)}$:

$$\rightarrow \left[\min_{x^{(1)},\dots,x^{(n_m)}} \boxed{\frac{1}{2}\sum_{\boxed{i=1}}^{n_m} \sum_{\boxed{j:r(i,j)=1}} \left((\theta^{(j)})^T x^{(i)} - y^{(i,j)}\right)^2} + \boxed{\frac{\lambda}{2}\sum_{i=1}^{n_m}\sum_{k=1}^{n}(x_k^{(i)})^2} \right. \leftarrow$$

Minimizing $x^{(1)},\dots,x^{(n_m)}$ and $\theta^{(1)},\dots,\theta^{(n_u)}$ simultaneously:

$$\underline{J(x^{(1)},\dots,x^{(n_m)},\boxed{\theta^{(1)},\dots,\theta^{(n_u)}})} = \boxed{\frac{1}{2}\sum_{(i,j):r(i,j)=1} \left((\theta^{(j)})^T x^{(i)} - y^{(i,j)}\right)^2} + \boxed{\frac{\lambda}{2}\sum_{i=1}^{n_m}\sum_{k=1}^{n}(x_k^{(i)})^2} + \boxed{\frac{\lambda}{2}\sum_{j=1}^{n_u}\sum_{k=1}^{n}(\theta_k^{(j)})^2}$$

$$\rightarrow \boxed{\min_{\substack{x^{(1)},\dots,x^{(n_m)} \\ \theta^{(1)},\dots,\theta^{(n_u)}}} J(x^{(1)},\dots,x^{(n_m)},\theta^{(1)},\dots,\theta^{(n_u)})}$$

$\theta \rightarrow x \rightarrow \theta \rightarrow x \rightarrow \dots$

Andrew Ng

$\cancel{x_0 = 1}$ $\quad x \in \mathbb{R}^n, \ \theta \in \mathbb{R}^n$

Collaborative filtering algorithm

1. Initialize $x^{(1)}, \ldots, x^{(n_m)}, \theta^{(1)}, \ldots, \theta^{(n_u)}$ to small random values.

$\cancel{\theta_0}$, θ_1, $\vdots$, θ_n

2. Minimize $J(x^{(1)}, \ldots, x^{(n_m)}, \theta^{(1)}, \ldots, \theta^{(n_u)})$ using gradient descent (or an advanced optimization algorithm). E.g. for every $j = 1, \ldots, n_u, i = 1, \ldots, n_m$:

$$x_k^{(i)} := x_k^{(i)} - \alpha \underbrace{\left(\sum_{j:r(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(i,j)})\theta_k^{(j)} + \lambda x_k^{(i)} \right)}_{\frac{\partial}{\partial x_k^{(i)}} J(\ldots)}$$

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

3. For a user with parameters θ and a movie with (learned) features x, predict a star rating of $\theta^T x$.

$(\theta^{(j)})^T x$

Extracting the right features from your data

- **Use explicit rating data**
- **Features needed**
  - **User IDs**
  - **Movie IDs**
  - **The ratings assigned to each (user, movie) pair**

Extracting the right features from your data

- **Extracting features from MovieLens 100k dataset**
  - ➢ *chg0901@ubuntu:~$ $SPARK_HOME/bin/spark-shell --driver-memory 4g*
  - ➢ *val rawData = sc.textFile("/home/chg0901/Desktop/ml-100k/u.data")*
    - **Ensure providing enough memory**

Extracting the right features from your data

▪ Extracting features from MovieLens 100k dataset

- **Inspect the raw ratings dataset**

➢ *rawData.first()*

```
scala> rawData.first()
res3: String = 196      242     3       881250949
```

- **this dataset consisted of** *user id, movie id, rating, ~~timestamp~~* **fields separated by a tab (*"\t"*) character.**

➢ *val rawRatings = rawData.map(_.split("\t").take(3))*

➢ *rawRatings.first()*

- **split each record on the "*\t*" character, which gives us an *Array[String]***
- **use Scala's *take* function to keep only the first 3 elements**
- ***first()* collect just the first record of *rawRatings* RDD**

```
scala> val rawRatings = rawData.map(_.split("\t").take(3))
rawRatings: org.apache.spark.rdd.RDD[Array[String]] = MappedRDD[4] at map at <co
nsole>:14

scala> rawRatings.first()
res4: Array[String] = Array(196, 242, 3)
```

Extracting the right features from your data

- **Extracting features from MovieLens 100k dataset**
  - **use Spark's *MLlib* library to train our model.**
  - **take a look at what methods are available for us to use and what input is required.**
  - **import the *ALS* model from *MLlib*:**

➢ ***import org.apache.spark.mllib.recommendation.ALS***

```
scala> import org.apache.spark.mllib.recommendation.ALS
import org.apache.spark.mllib.recommendation.ALS

scala> ALS.
analyzeBlocks   asInstanceOf    isInstanceOf    toString        train
trainImplicit

scala> ALS.train
<console>:12: error: ambiguous reference to overloaded definition,
both method train in object ALS of type (ratings: org.apache.spark.rdd.RDD[org.a
pache.spark.mllib.recommendation.Rating], rank: Int, iterations: Int)org.apache.
spark.mllib.recommendation.MatrixFactorizationModel
and  method train in object ALS of type (ratings: org.apache.spark.rdd.RDD[org.a
pache.spark.mllib.recommendation.Rating], rank: Int, iterations: Int, lambda: Do
uble)org.apache.spark.mllib.recommendation.MatrixFactorizationModel
match expected type ?
              ALS.train
                  ^
```

Extracting the right features from your data

- **Extracting features from MovieLens 100k dataset**
  - **We need another three inputs**
    - ***rank*: number of factors in the *ALS* model(number of hidden features)**
      - **Bigger then better, but memory usage(computation and store model)**
      - **Reasonable range : 10 ~ 200 [*50*]**
    - ***iterations*: number of iterations to run**
      - **Each iterations is guaranteed to decrease the reconstruction error**
      - **Converge quickly: around 10 [*10*]**
    - ***lambda*: control the regularization of model → ctrl over fitting**
      - **Higher then more**
      - **Dependent to the size, nature, and sparsity of data [*0.01*]**
      - **Working with the cross-validation approaches**

Extracting the right features from your data

- **Extracting features from MovieLens 100k dataset**
  - *import org.apache.spark.mllib.recommendation.Rating*

```
scala> import org.apache.spark.mllib.recommendation.Rating
import org.apache.spark.mllib.recommendation.Rating

scala> Rating.
apply             asInstanceOf    curried        isInstanceOf    toString
tupled            unapply

scala> Rating()
<console>:13: error: not enough arguments for method apply: (user: Int, product:
 Int, rating: Double)org.apache.spark.mllib.recommendation.Rating in object Rati
ng.
Unspecified value parameters user, product, rating.
              Rating()
                    ^
```

- **we need to provide the *ALS* model with an *RDD* that consists of *Rating* records.**
- **A *Rating* class is just a wrapper around *user id, movie id* (*product*), and the actual *rating* arguments.**

Extracting the right features from your data

- **Extracting features from MovieLens 100k dataset**
  - **Create our rating dataset using the *map* method and transforming the array of IDs(user and movie) and ratings into a *Rating* object**

  ➢ *val ratings = rawRatings.map { case Array(user, movie, rating) => Rating(user.toInt, movie.toInt, rating.toDouble) }*

  ➢ *ratings.first()*

```
scala> val ratings = rawRatings.map { case Array(user, movie, rating) => Rating(
user.toInt, movie.toInt, rating.toDouble) }
ratings: org.apache.spark.rdd.RDD[org.apache.spark.mllib.recommendation.Rating]
= MappedRDD[3] at map at <console>:18

scala> ratings.first()
res3: org.apache.spark.mllib.recommendation.Rating = Rating(196,242,3.0)
```

  - ***toInt toDouble* convert string raw rating data**
  - ***case* extract the relevant variable names and use them directly (this saves us from having to use something like *val user = ratings(0)*).**
    - **Case pattern matching:** http://docs.scala-lang.org/tutorials/tour/pattern-matching.html

Training the recommendation model

- **Training a model using explicit data (*train*)**
  - ➢ *val model = ALS.train(ratings, 50, 10, 0.01)*
    - use *rank* of *50*, *10 iterations*, and a *lambda* parameter of *0.01* to illustrate how to train our model
    - Returns a *MatrixFactorizationModel* object, which contains the user and item factors in the form of an RDD of (*id, factor*) pairs called *userFeatures* and *productFeatures*.

```
scala> val model = ALS.train(ratings, 50, 10, 0.01)
17/03/19 10:40:42 WARN MatrixFactorizationModel: User factor does not have a par
titioner. Prediction on individual records could be slow.
17/03/19 10:40:42 WARN MatrixFactorizationModel: Product factor does not have a
partitioner. Prediction on individual records could be slow.
model: org.apache.spark.mllib.recommendation.MatrixFactorizationModel = org.apac
he.spark.mllib.recommendation.MatrixFactorizationModel@4d86f330

scala> model.userFeatures
res4: org.apache.spark.rdd.RDD[(Int, Array[Double])] = usersOut FlatMappedRDD[39
9] at flatMap at ALS.scala:393
```

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

Training the recommendation model

- **Training a model using explicit data (*train*)**
  - **Operations used in *MLlib's* *ALS* implementation are lazy transformations, so actual computation will only be performed once we call some sort of action on the resulting RDDs**
  - **We can force the computation using a *Spark* action such as *count*:**

➢ *model.userFeatures.count*

➢ *model.productFeatures.count*

```
scala> model.userFeatures.count
res5: Long = 943

scala> model.productFeatures.count
res6: Long = 1682
```

  - **As expected, we have a factor array for each user (*943* factors) and movie (*1682* factors).**
  - **The standard matrix factorization approach in *MLlib* deals with explicit ratings like the *train***

Training the recommendation model

- **Training a model using implicit feedback data**
  - **To work with implicit data, you can use the *`trainImplicit`* method.**
  - **additional parameter : *`alpha`***
    - **controls the baseline level of confidence weighting applied.**
    - **A higher level of *`alpha`* tends to make the model more confident about the fact that missing data equates to no preference for the relevant user-item pair.**
  - **Convert MovieLens dataset into an implicit dataset**
    - **convert into binary feedback (0s and 1s) by applying a threshold on the ratings at some level.**
    - **convert the ratings' values into confidence weights**

Training the recommendation model

- **Training a model using implicit feedback data**

```
scala> val model2 = ALS.trainImplicit



def trainImplicit(ratings: org.apache.spark.rdd.RDD[org.apache.spark.mllib.recom
mendation.Rating], rank: Int, iterations: Int): MatrixFactorizationModel

def trainImplicit(ratings: org.apache.spark.rdd.RDD[org.apache.spark.mllib.recom
mendation.Rating], rank: Int, iterations: Int, lambda: Double, alpha: Double): M
atrixFactorizationModel
def trainImplicit(ratings: org.apache.spark.rdd.RDD[org.apache.spark.mllib.recom
mendation.Rating], rank: Int, iterations: Int, lambda: Double, blocks: Int, alph
a: Double): MatrixFactorizationModel
def trainImplicit(ratings: org.apache.spark.rdd.RDD[org.apache.spark.mllib.recom
mendation.Rating], rank: Int, iterations: Int, lambda: Double, blocks: Int, alph
a: Double, seed: Long): MatrixFactorizationModel

scala> ratings
res6: org.apache.spark.rdd.RDD[org.apache.spark.mllib.recommendation.Rating] = M
appedRDD[3] at map at <console>:18

scala> val model2 = ALS.trainImplicit(ratings,50,10,0.01,0.5)
17/03/19 22:08:40 WARN MatrixFactorizationModel: User factor does not have a par
titioner. Prediction on individual records could be slow.
17/03/19 22:08:40 WARN MatrixFactorizationModel: Product factor does not have a
partitioner. Prediction on individual records could be slow.
model2: org.apache.spark.mllib.recommendation.MatrixFactorizationModel = org.apa
che.spark.mllib.recommendation.MatrixFactorizationModel@6183dd2
```

Using the recommendation model

- **To make predictions**
  - Recommendation for a given user
  - Recommendation for related or similar items for a given item
- **User recommendation**
  - Generate recommended items for a given user
    - Top-K list: the K items with the highest probability of user liking them
    - By computing the predicted score for each item and ranking the list
  - Method to perform the computation
    - User-based: the ratings of similar users on items are used to compute
    - Item-based: the similarity of items the user has rated to the candidate items
  - Compute score with matrix factorization as
    - vector dot product between a user-factor vector and an item-factor vector

Using the recommendation model

- **Generating movie recommendations**
  - **The recommendation model is based on matrix factorization**
  - **use the factor matrices to compute predicted scores (or ratings) for a user**
  - **The *MatrixFactorizationModel* class has a convenient predict method to compute a predicted score**

➢ *val predictedRating = model.predict(789, 123)*

```
scala> val predictedRating = model.predict(789, 123)
predictedRating: Double = 3.9404331107411106
```

  - **It predicts a rating of *3.9* for user *789* and movie *123***
  - **The *predict* method can take *RDD* (*user, item*) pairs as input, then generate predictions**

Using the recommendation model

- **Generating movie recommendations**
  - **To generate the top-*K* recommended items for a user**
  - ***MatricxFeactorizationModel* provides a convenience method called *recommendProducts***
    - **Two arguments: *user* and *items***
    - **Return the top *num* items ranked in the order of the predicted score**
  - **Generate top-*10* recommended items for user *789***

➢ *val userId = 789 ; val K = 10*
➢ *val topKRecs = model.recommendProducts(userId, K)*
➢ *println(topKRecs.mkString("\n"))*

```
scala> println(topKRecs.mkString("\n"))
Rating(789,959,5.741979784406875)
Rating(789,675,5.669828486058768)
Rating(789,661,5.355355074994423)
Rating(789,528,5.343261387246371)
Rating(789,573,5.30831263582184)
Rating(789,429,5.286993066653898)
Rating(789,530,5.204722334235818)
Rating(789,430,5.1974465187061245)
Rating(789,484,5.097355446131142)
Rating(789,302,5.079177393321885)
```

Using the recommendation model

- **Inspecting the recommendations**
  - **A sense check**
    - **Load the movie data**
    - **Collect the data as *Map[Int, String]* method mapping the movie ID to title**

➢ *val movies = sc.textFile("/home/chg0901/Desktop/ml-100k/u.item")*

```
u.item     -- Information about the items (movies); this is a tab separated
              list of
              movie id | movie title | release date | video release date |
              IMDb URL | unknown | Action | Adventure | Animation |
              Children's | Comedy | Crime | Documentary | Drama | Fantasy |
              Film-Noir | Horror | Musical | Mystery | Romance | Sci-Fi |
              Thriller | War | Western |
              The last 19 fields are the genres, a 1 indicates the movie
              is of that genre, a 0 indicates it is not; movies can be in
              several genres at once.
              The movie ids are the ones used in the u.data data set.
```

```
scala> movies.first()
res8: String = 1|Toy Story (1995)|01-Jan-1995||http://us.imdb.com/M/title-exact?
Toy%20Story%20(1995)|0|0|0|1|1|1|0|0|0|0|0|0|0|0|0|0|0|0|0
```

➢ *val titles = movies.map(line => line.split("\\|").take(2)).map(array => (array(0).toInt, array(1))).collectAsMap()*

➢ *titles(1)*

```
scala> titles(1)
res53: String = Toy Story (1995)
```

Using the recommendation model

Inspecting the recommendations

- **For user *789***
  - We can find out what movies they have rated
  - Take the *10* movies with highest rating
  - Then check the titles
- **Using the *keyBy* *Spark* function to create an *RDD* of key-value pairs from our ratings *RDD***
  - Use the *lookup* function to return just ratings for this key(*789*)
  - Then see how many movies this user(*789*) has rated
    - by the *size* of the *movieForUser* collection

➢ *val moviesForUser = ratings.keyBy(_.user).lookup(789)*
➢ *println(moviesForUser.size)*

```
scala> val moviesForUser = ratings.keyBy(_.user).lookup(789)
moviesForUser: Seq[org.apache.spark.mllib.recommendation.Rating] = WrappedArray(
Rating(789,1012,4.0), Rating(789,127,5.0), Rating(789,475,5.0), Rating(789,93,4.
0), Rating(789,1161,3.0), Rating(789,286,1.0), Rating(789,293,4.0), Rating(789,9
,5.0), Rating(789,50,5.0), Rating(789,294,3.0), Rating(789,181,4.0), Rating(789,
1,3.0), Rating(789,1008,4.0), Rating(789,508,4.0), Rating(789,284,3.0), Rating(7
89,1017,3.0), Rating(789,137,2.0), Rating(789,111,3.0), Rating(789,742,3.0), Rat
ing(789,248,3.0), Rating(789,249,3.0), Rating(789,1007,4.0), Rating(789,591,3.0)
, Rating(789,150,5.0), Rating(789,276,5.0), Rating(789,151,2.0), Rating(789,129,
5.0), Rating(789,100,5.0), Rating(789,741,5.0), Rating(789,288,3.0), Rating(789,
762,3.0), Rating(789,628,3.0), Rating(789,124,4.0))

scala> println(moviesForUser.size)
33
```

Using the recommendation model

- **Inspecting the recommendations**
  - **Take the *10* movies with the highest rating**
    - By sorting the *moviesForUser* collection using the 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
    - Print out the top *10* titles with their ratings

?

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

```
scala> moviesForUser.sortBy(-_.rating).take(10).map(rating => (titles(rating.pro
duct), 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)
```

sortBy with the "*-*"

- *sortBy* with the "*-*" or not
  - With it, the rating is descending
  - Without it, the rating is increasing

```
scala> moviesForUser.sortBy(-_.rating).take(10).map(rating => (titles(rating.pro
duct), 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)

scala> moviesForUser.sortBy(_.rating).take(10).map(rating => (titles(rating.prod
uct), rating.rating)).foreach(println)
(English Patient, The (1996),1.0)
(Big Night (1996),2.0)
(Willy Wonka and the Chocolate Factory (1971),2.0)
(Palookaville (1996),3.0)
(Liar Liar (1997),3.0)
(Toy Story (1995),3.0)
(Tin Cup (1996),3.0)
(Trees Lounge (1996),3.0)
(Truth About Cats & Dogs, The (1996),3.0)
(Ransom (1996),3.0)
```

Using the recommendation model

- **Inspecting the recommendations**
  - **Top *10* recommendations for user *789***
    - **See what the titles are using the same approach as the one we used earlier**

➢ *topKRecs.map(rating => (titles(rating.product), rating.rating)).foreach(println)*

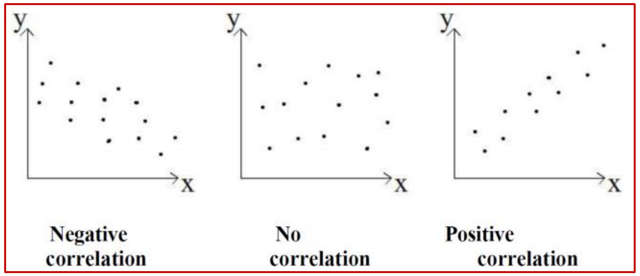
```
scala> topKRecs.map(rating => (titles(rating.product), rating.rating)).foreach(p
rintln)
(Perfect World, A (1993),5.617116590577549)
(Raging Bull (1980),5.576513746732209)
(Jackie Brown (1997),5.568204439222325)
(Searching for Bobby Fischer (1993),5.426807430223894)
(Nosferatu (Nosferatu, eine Symphonie des Grauens) (1922),5.353372579981078)
(Big Sleep, The (1946),5.291525230118406)
(My Man Godfrey (1936),5.267526356704597)
(Hoop Dreams (1994),5.255371890802268)
(Belle de jour (1967),5.241991938070711)
(Fantasia (1940),5.224851903662856)
```

- These recommendations make sense ?

Using the recommendation model

- **Item recommendations**
  - **For a certain item, find the most similar items**
  - **Definitions of similarity**
    - **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**
    - **Pearson correlation**
    - **Cosine similarity for real-value vectors**
    - **Jaccard similarity for binary vectors**

Using the recommendation model



- **Item recommendations**
  - **Common similarity measures**
    - **Pearson correlation coefficient**
      - **the covariance of the two variables divided by the product of their standard deviations.**

For a population

$$p(x,y) = \frac{\sum x_i y_i - n\bar{x}\bar{y}}{(n-1)s_x s_y} = \frac{n\sum x_i y_i - \sum x_i \sum y_i}{\sqrt{n\sum x_i^2 - (\sum x_i)^2}\sqrt{n\sum y_i^2 - (\sum y_i)^2}}$$

Pearson's correlation coefficient when applied to a population is commonly represented by the Greek letter ρ (rho) and may be referred to as the *population correlation coefficient* or the *population Pearson correlation coefficient*. The formula for ρ[7] is:

$$\rho_{X,Y} = \frac{\operatorname{cov}(X,Y)}{\sigma_X \sigma_Y}$$

where:

- cov is the covariance
- σ_X is the standard deviation of X
- σ_Y is the standard deviation of Y

Using the recommendation model

Item recommendations

Common similarity measures

- **Pearson correlation coefficient**
  - **the covariance of the two variables divided by the product of their standard deviations.**

The formula for ρ can be expressed in terms of uncentered moments. Since

- $\mu_X = \mathrm{E}[X]$
- $\mu_Y = \mathrm{E}[Y]$
- $\sigma_X^2 = \mathrm{E}[(X - \mathrm{E}[X])^2] = \mathrm{E}[X^2] - [\,E[X]]^2$
- $\sigma_Y^2 = \mathrm{E}[(Y - \mathrm{E}[Y])^2] = \mathrm{E}[Y^2] - [\,E[Y]]^2$
- $$\mathrm{E}[(X - \mu_X)(Y - \mu_Y)] = \mathrm{E}[(X - \mathrm{E}[X])(Y - \mathrm{E}[Y])] = \mathrm{E}[XY] - \mathrm{E}[X]\,\mathrm{E}[Y],$$

the formula for ρ can also be written as

$$\rho_{X,Y} = \frac{\mathrm{E}[XY] - \mathrm{E}[X]\,\mathrm{E}[Y]}{\sqrt{\mathrm{E}[X^2] - [\,E[X]]^2}\,\sqrt{\mathrm{E}[Y^2] - [\,E[Y]]^2}}.$$

Using the recommendation model

- **Item recommendations**
  - **Common similarity measures**
    - **Cosine similarity for real-value vectors**

$$T(x,y) = \frac{x \bullet y}{\|x\|^2 \times \|y\|^2} = \frac{\sum x_i y_i}{\sqrt{\sum x_i^2}\sqrt{\sum y_i^2}}$$

    - **Jaccard similarity for binary vectors(**Tanimoto Coefficient**)**

$$T(x,y) = \frac{x \bullet y}{\|x\|^2 + \|y\|^2 - x \bullet y} = \frac{\sum x_i y_i}{\sqrt{\sum x_i^2} + \sqrt{\sum y_i^2} - \sum x_i y_i}$$

    - **Euclidean distance**

$$d(x,y) = \sqrt{(\sum (x_i - y_i)^2)} \qquad sim(x,y) = \frac{1}{1 + d(x,y)}$$

Collaborative filtering

- **[user-based]**
  - **If we assume there are two users *u* and *v***
  - ***N(u), N(v)* represents the item set that user *u, v* have shown a preference for, respectively.**
  - **Then we have two method to calculator the similarity:**
    - **Jaccard similarity** $$w_{uv} = \frac{|N(u) \cap N(v)|}{|N(u) \cup N(v)|}$$
    - **(Cosine similarity)** $$w_{uv} = \frac{|N(u) \cap N(v)|}{\sqrt{|N(u)||N(v)|}}$$
  - **Example**
    - **User A item set: {a, b, d}, user B item set: {a, c, f}**

$$w_{AB} = \frac{|\{a, b, d\} \bigcap \{a, c, f\}|}{\sqrt{|\{a, b, d\}||\{a, c, f\}|}} = \frac{|1|}{\sqrt{|3||3|}} = \frac{1}{3}$$

ItemCF AND IUF

- **ItemCF**
  - **the similarity of the items *i* and *j* :**
- $$w_{ij} = \frac{|N(i) \cap N(j)|}{|N(i)|} \quad \text{or} \quad w_{ij} = \frac{|N(i) \cap N(j)|}{\sqrt{|N(i)||N(j)|}}$$
  - **$|N(i)|$ is the number of users who likes item *i*.**
  - **$|N(i) \cap N(j)|$ is the number of users who likes *i* and *j*.**
  - **For the first formula, if item *j* is very popular, almost everyone have seen that, in this case, the W_{ij} close to *1*.**
    - **To avoid this case, we use the second formula as usual.**
- **ItemCF-IUF(Inverse User Frequence)**
  - **$N(u)$ is list of preference items of user *u***
  - **The IUF of user *u* is** $\frac{1}{log(1+|N(u)|)}$
  - **Similarity of the items *A* and *B* :**

$$w_{AB} = \frac{\sum_{u \in N(A) \cap N(B)} \frac{1}{log(1+|N(U)|)}}{\sqrt{|N(A)||N(B)|}}$$

Using the recommendation model

- **Item recommendations**
  - **Generating similar movies**
    - **Compute the required vector dot products**
      - **Use the cosine similarity metric**
      - **Use the jblas (basic)linear algebra library(for Java , http://jblas.org/)**
    - **compare the factor vector of our chosen item with each of the other items, using our similarity metric and to perform linear algebra computations,**
      - **first create a vector object out of the factor vectors, which are in the form of an *Array[Double]*.**
      - **The JBLAS class, *DoubleMatrix*, takes an *Array[Double]* as the constructor argument as follows:**

➢ *import org.jblas.DoubleMatrix*
➢ *val aMatrix = new DoubleMatrix(Array(1.0, 2.0, 3.0))*

```
scala> import org.jblas.DoubleMatrix
import org.jblas.DoubleMatrix

scala> val aMatrix = new DoubleMatrix(Array(1.0, 2.0, 3.0))
aMatrix: org.jblas.DoubleMatrix = [1.000000; 2.000000; 3.000000]
```

Using the recommendation model

- **Generating similar movies**
  - **Compute the cosine similarity between two vectors**
    - **Cosine similarity** is a measure of the angle between two vectors in an n-dimensional space.
    - calculating the dot product between the vectors
    - dividing the result by a denominator, which is the (L2-)norm (or length) of each vector multiplied together.
      - In this way, cosine similarity is a normalized dot product.

```
import org.jblas.DoubleMatrix
val aMatrix = new DoubleMatrix(Array(1.0, 2.0, 3.0))
```

  - **The cosine similarity measure takes on values between *-1* and *1*.**
    - A value of 1 implies completely similar
    - A value of 0 implies independence (no similarity)
    - A value of -1 implies that not only are the vectors not similar, but they are also completely dissimilar（captures negative similarity）

Using the recommendation model

▪ Generating similar movies

- **Create the cosineSimularity function**

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

- **Test with item *567***
  - Collect an item factor from our model using *lookup* method
  - Use the *head* function
    - Since the *lookup* returns an array of values
    - we just need the first one value or just one value and it's the factor vector for the item
  - Compute the cosine similarity with the item *567* itself
    - *Array[Double]* ← create a *DoubleMatrix* object

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

```
scala> cosineSimilarity(itemVector, itemVector)
res19: Double = 0.9999999999999999
```

the item factor is identical to itself

Using the recommendation model

- **Generating similar movies**
  - Apply the similarity metric to each item

```
val sims = model.productFeatures.map{ case (id, factor) =>
    val factorVector = new DoubleMatrix(factor)
    val sim = cosineSimilarity(factorVector, itemVector)
    (id, sim)
  }
```

  - Compute the top *10* most similar items by sorting out the similarity score for each item

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

  - Spark's *top* function
    - An efficient way to compute top-*K* result in a distributed fashion
    - Instead of using *collect* to return all the data to the driver and sorting it locally
  - How to sort (*item id, similarity score*) pairs in *sims* RDD
    - Pass an extra argument to top which is a Scala *Ordering* object that tell Spark that is should sort by the value(*similarity*) in the key-value pair

Using the recommendation model

Generating similar movies

- **Finally, print the *10* items with the highest computed similarity metric to the given item(*567*)**
  - ➢*println(sortedSims.mkString("\n"))*
- **The top-ranked similar item is our item(*567*)**
- **The rest are the other items in our set of items, ranked in order of our similarity metric**

```
scala> println(sortedSims.mkString("\n"))
(567,0.9999999999999999)
(413,0.6762576692149287)
(288,0.6616028077638827)
(219,0.660845989200629)
(853,0.6580347448933894)
(232,0.6550624221907995)
(430,0.645533441720036)
(563,0.6438632998576848)
(1505,0.6424377105789582)
(859,0.6423957711821132)
```

Using the recommendation model

- **Item Recommendations**
  - **Inspecting the similar items**
    - The title of our chosen movie is
    - ➢ *println(titles(itemId))*
  - **Sense check**
    - Titles of the most similar movies
    - Item to item similarity computations
    - Take the top *11* → take the numbers *1* to *11* in the list
    - ➢ *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")*

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

```
scala> val sortedSims2 = sims.top(K + 1)(Ordering.by[(Int, Double), Dou
ble] { case (id, similarity) => similarity })
sortedSims2: Array[(Int, Double)] = Array((567,0.9999999999999999), (41
3,0.6762576692149287), (288,0.6616028077638827), (219,0.660845989200629
), (853,0.6580347448933894), (232,0.6550624221907995), (430,0.645533344
1720036), (563,0.6438632998576848), (1505,0.6424377105789582), (859,0.6
423957711821132), (1083,0.6408537664724357))

scala> sortedSims2.slice(1, 11).map{ case (id, sim) => (titles(id), sim
) }.mkString("\n")
res26: String =
(Tales from the Crypt Presents: Bordello of Blood (1996),0.676257669214
9287)
(Scream (1996),0.6616028077638827)
(Nightmare on Elm Street, A (1984),0.660845989200629)
(Braindead (1992),0.6580347448933894)
(Young Guns (1988),0.6550624221907995)
(Duck Soup (1933),0.6455333441720036)
(Stephen King's The Langoliers (1995),0.6438632998576848)
(Killer: A Journal of Murder (1995),0.6424377105789582)
(April Fool's Day (1986),0.6423957711821132)
(Albino Alligator (1996),0.6408537664724357)
```

Evaluating the performance of models

- Whether the model is a good model?
  - To evaluate its predictive performance
  - Direct measures
    - How well a model predicts model's target variable(MSE)
    - How well a model performs at predicting things (MAP)
      - Might not be directly optimized in the model
      - But are often closer to what we care about in the real world
  - Evaluation metrics
    - Measures of a model's predictive capability or accuracy
    - Provide a standardized way of
      - Comparing the performance of the same model with different parameter settings
      - Comparing performance across different models
    - Two common evaluation metrics used in recommender systems and collaborative filtering models
      - Mean squared error (MSE)
      - Mean average precision (MAP) at K → MAPK
  - Using Mllib's built-in evaluation functions

Evaluating the performance of models

- **Mean Squared Error**
  - **A direct measure of the reconstruction error of the user-item rating matrix**
    - **the objective function being minimized in certain models, specifically many matrix-factorization techniques, including ALS.**
    - **commonly used in explicit ratings settings**
  - **Definition**
    - **the sum of the squared errors divided by the number of observations.**
    - **the square of the difference between the predicted rating for a given user-item pair and the actual rating.**
  - **Example → user *789***
    - **Take the first rating for this user from the movieForUser set of Rating**

➢ *val actualRating = moviesForUser.take(1)(0)*

```
scala> val actualRating = moviesForUser.take(1)(0)
actualRating: org.apache.spark.mllib.recommendation.Rating
= Rating(789,1012,4.0)
```

Evaluating the performance of models

- **Mean Squared Error**
  - **Compute the model's predicted rating**

➢ *val predictedRating = model.predict(789, actualRating.product)*

```
scala> val actualRating = moviesForUser.take(1)(0)
actualRating: org.apache.spark.mllib.recommendation.Rating = Rating(78
,1012,4.0)

scala> val predictedRating = model.predict(789, actualRating.product)
predictedRating: Double = 3.99106505239374
```

  - **predicted rating is *3.99*, very close to the actual rating**
  - **Compute the squared error between the actual and predict rating**

➢ *val squaredError = math.pow(predictedRating - actualRating.rating, 2.0)*

```
scala> val squaredError = math.pow(predictedRating - actualRating.ratin
g, 2.0)
squaredError: Double = 7.983328872661352E-5
```

Evaluating the performance of models

- **Mean Squared Error**
  - **To compute the overall MSE for the dataset**
    - Compute each squared error for each (*user, movie, actual rating, predict rating*) entry → *sum* them up → divide them by the number of *ratings*

```
➢val usersProducts = ratings.map{ case Rating(user, product, rating)  =>
 (user, product)}
➢val predictions = model.predict(usersProducts).map{
      case Rating(user, product, rating) => ((user, product), rating)
 }
```

  - Extract user and product IDs from the ratings RDD
  - Make predictions for each user-item pair using model.predict
  - Use user-item pair as the key and the predicted rating as the value

```
scala> val usersProducts = ratings.map{ case Rating(user, product, rating)  =>
 (user, product)}
usersProducts: org.apache.spark.rdd.RDD[(Int, Int)] = MappedRDD[256] at map at
 <console>:29

scala> val predictions = model.predict(usersProducts).map{
     |     case Rating(user, product, rating) => ((user, product), rating)
     | }
predictions: org.apache.spark.rdd.RDD[((Int, Int), Double)] = MappedRDD[265] a
t map at <console>:33
```

Evaluating the performance of models

- **Mean Squared Error**
  - **extract the actual ratings and *`map`* the ratings RDD**
    - **so that the user-item pair is the key and the *`actual rating`* is the value**
  - ***`join`* two RDDs with the same form of key to create a new RDD**
    - **the actual and *`predicted ratings`* for each *`user-item`* combination**

```
➢val ratingsAndPredictions = ratings.map{
    case Rating(user, product, rating) => ((user, product), rating)
 }.join(predictions)
```

- **Get the (*`user, movie, actual rating, predict rating`*) entry**

```
scala> val ratingsAndPredictions = ratings.map{
     |     case Rating(user, product, rating) => ((user, product), rating)
     | }.join(predictions)
ratingsAndPredictions: org.apache.spark.rdd.RDD[((Int, Int), (Double, Double))
] = FlatMappedValuesRDD[269] at join at <console>:37
```

Evaluating the performance of models

- **Mean Squared Error**
  - **Compute the MSE**
    - **summing up the squared errors using reduce**
    - **dividing by the count method of the number of records**

```
➢ val MSE = ratingsAndPredictions.map{
    case ((user, product),(actual, predicted)) => math.pow((actual - predicted), 2)
  }.reduce(_ + _) / ratingsAndPredictions.count
➢ println("Mean Squared Error = " + MSE)
```

```
scala> println("Mean Squared Error = " + MSE)
Mean Squared Error = 0.0832917627824916
```

  - **Root Mean Squared Error（RMSE)**
    - **Squared root of MSE, more common to use**
    - **Somewhat more interpretable**
    - **equivalent to the standard deviation of the differences between the predicted and actual ratings.**

```
➢ val RMSE = math.sqrt(MSE)
➢ println("Root Mean Squared Error = " + RMSE)
```

➢ Output

```
scala> println("Root Mean Squared Error = " + RMSE)
Root Mean Squared Error = 0.2886031233068894
```

Evaluating the performance of models

- Average precision at K (APK)
  - Mean average precision at K (MAPK)
    - The mean of average precision at K metric across all instance in the dataset
  - APK is a measure of the average relevance scores of a set of the top-K documents presented in response to a query
  - For each query instance, we will compare the set of top-K results with the set of actual relevant documents
  - More appropriate to evaluate the implicit datasets
- Evaluate model with APK
  - each user is the equivalent of a query
  - the set of top-K recommended items is the document result set
  - The relevant documents is the set of items that a user interacted with.
  - APK attempts to measure how good our model is at predicting items that a user will find relevant and choose to interact with.

Evaluating the performance of models

- **Evaluate model with APK**
  - **Function to compute the APK**

```scala
def avgPrecisionK(actual: Seq[Int], predicted: Seq[Int], k: Int): Double = {
  val predK = predicted.take(k)
  var score = 0.0
  var numHits = 0.0
  for ((p, i) <- predK.zipWithIndex) {
    if (actual.contains(p)) {
      numHits += 1.0
      score += numHits / (i.toDouble + 1.0)
    }
  }
  if (actual.isEmpty) {
    1.0
  } else {
    score / scala.math.min(actual.size, k).toDouble
  }
}
```

  - **our estimate will be relevant for the user**
    - **takes as input a list of actual item IDs that are associated with the user and another list of predicted ids**
  - ***zipWithIndex***
    - **This function takes an RDD of values and merges them together with an index to create a new RDD of key-value pairs, where the key will be the term and the value will be the index in the term dictionary.**

Evaluating the performance of models

- **Evaluate model with APK**
  - **Example : compute APK metric for user *789***
    - **Extract the actual movie IDs for the user**

➢ *val actualMovies = moviesForUser.map(_.product)*

```
scala> val actualMovies = moviesForUser.map(_.product)
actualMovies: Seq[Int] = ArrayBuffer(1012, 127, 475, 93, 1161, 286, 293, 9, 50
, 294, 181, 1, 1008, 508, 284, 1017, 137, 111, 742, 248, 249, 1007, 591, 150,
276, 151, 129, 100, 741, 288, 762, 628, 124)
```

    - **Use movie recommendations to compute the APK score using K = *10***

➢ *val predictedMovies = topKRecs.map(_.product)*

```
scala> val predictedMovies = topKRecs.map(_.product)
predictedMovies: Array[Int] = Array(447, 429, 32, 179, 211, 96, 474, 675, 56, 526)
```

    - **Produce the average precision**

➢ *val apk10 = avgPrecisionK(actualMovies, predictedMovies, 10)*

```
scala> val apk10 = avgPrecisionK(actualMovies, predictedMovies, 10)
apk10: Double = 0.0
```

- **In this case, we can see that our model is not doing a very good job of predicting relevant movies for user *789* as APK score is *0***

Evaluating the performance of models

- **Compute the overall MAPK**
  - **Compute the APK for every user and average them**
    - **Generate the list of recommendations for each user in our dataset**
      - Fairly intensive on a large scale →
      - Distribute the computation using Spark functionality
    - **Limitation**: each worker must have the full item-factor matrix available
      - it can compute dot product between relevant user vector and all item vector
      - This can be a problem when the number of items is extremely high as the item matrix must fit in the memory of one machine.
    - **Collect the item factors and form a *DoubleMatrix* object from them**

```
val itemFactors = model.productFeatures.map { case (id, factor) => factor }.collect()
val itemMatrix = new DoubleMatrix(itemFactors)
println(itemMatrix.rows, itemMatrix.columns)
```

```
scala> println(itemMatrix.rows, itemMatrix.columns)
(1682,50)
```

      - *itemMatrix* is a matrix with *1682* rows and *50* columns
      - The number of movies and factor dimension

Evaluating the performance of models

- **Compute the overall MAPK**
  - **Distribute the item matrix as a *broadcast* variable**
    - **So that it is available on each worker node**

➢ *val imBroadcast = sc.broadcast(itemMatrix)*

```
scala> val imBroadcast = sc.broadcast(itemMatrix)
imBroadcast: org.apache.spark.broadcast.Broadcast[org.jblas.DoubleMatrix]
 = Broadcast(60)
```

  - **Ready to compute the recommendations for each user**
    - **Apply a *map* function to each user**
      - **perform a matrix multiplication between user-factor vector and movie-factor matrix**
    - **The result is a vector with the predicted rating for each movie**
      - **The length of the vector is *1682*, which is the number of movies**

➢
```
val allRecs = model.userFeatures.map{ case (userId, array) =>
    val userVector = new DoubleMatrix(array)
    val scores = imBroadcast.value.mmul(userVector)
    val sortedWithId = scores.data.zipWithIndex.sortBy(-_._1)
    val recommendedIds = sortedWithId.map(_._2 + 1).toSeq
    (userId, recommendedIds)
  }
```

```
allRecs: org.apache.spark.rdd.RDD[(Int, Seq[Int])] = MappedRDD[272] at ma
p at <console>:39
```

    - **The RDD contains a list of movie IDs for each user ID**
      - **These movies IDs are sorted in order of the estimated rating**
      - **The add-1 : item-factor matrix is 0-indexed while movie IDs starts at 1**

Evaluating the performance of models

- **Compute the overall MAPK**
  - **Need the list of movie IDs for each user to pass into APK function as actual argument**
    - **Extract the user and movie IDs from the ratings RDD**
    - **Use Spark's *groupBy* operator**
      - **Get an RDD that contains a list of (userId, movieId) pairs for each user ID (key)**

➢ *val userMovies = ratings.map{ case Rating(user, product, rating) => (user, product) }.groupBy(_._1) imBroadcast = sc.broadcast(itemMatrix)*

```
scala> val userMovies = ratings.map{ case Rating(user, product, rating) =
> (user, product) }.groupBy(_._1)
userMovies: org.apache.spark.rdd.RDD[(Int, Iterable[(Int, Int)])] = Shuff
ledRDD[275] at groupBy at <console>:29
```

Evaluating the performance of models

- **Compute the overall MAPK**
  - **Use Spark's join operator to join these two RDDs together on the user ID key**
    - **For each user, we have the list of actual and predicted movie IDs that we can pass to our APK function**
    - **Sum each of these APK scores using a reduce action**
    - **Divide by the number of users (count of allRecs RDD)**

```
➢ val K = 10
➢ val MAPK = allRecs.join(userMovies).map{ case (userId, (predicted, actualWithIds)) =>
    val actual = actualWithIds.map(_._2).toSeq
    avgPrecisionK(actual, predicted, K)
  }.reduce(_ + _) / allRecs.count
➢ println("Mean Average Precision at K = " + MAPK)
```

```
scala> println("Mean Average Precision at K = " + MAPK)
Mean Average Precision at K = 0.05505251729535936
```

    - **The model achieves a fairly low MAPK**
    - **Typical values for recommendations tasks are usually relatively low**
      - **Especially if the item set is extremely large**
- **Try different parameters(lambda, rank or alpha)?**

Evaluating the performance of models

- **Using Mllib's built-in evaluation functions**
  - **MSE/RMSE and MAPK from scratch**
  - **Mllib provides convenience functions for evaluation**
    - ***RegressionMetrics*** **and** ***RankingMetrics*** **classes**
  - **RMSE and MSE**
    - **instantiate a** ***RegressionMetrics*** **instance by passing in an RDD of key-value pairs that represent the predicted and true values for each data point**

```
➢ import org.apache.spark.mllib.evaluation.RegressionMetrics
➢ val predictedAndTrue = ratingsAndPredictions.map { case ((user, product), (actual, predicted)) => (actual, predicted) }
➢ val regressionMetrics = new RegressionMetrics(predictedAndTrue)
➢ println("Mean Squared Error = " + regressionMetrics.meanSquaredError)
➢ println("Root Mean Squared Error = " + regressionMetrics.rootMeanSquaredError)
```

- **The output is exactly the same as we computed earlier**

```
scala> println("Mean Squared Error = " + regressionMetrics.meanSquaredError)
Mean Squared Error = 0.08329176278249159

scala> println("Root Mean Squared Error = " + regressionMetrics.rootMeanSquaredError)

Root Mean Squared Error = 0.28860312330688936
```

Evaluating the performance of models

- **Using Mllib's built-in evaluation functions**
  - **MAP**
    - **compute ranking-based evaluation metrics using MLlib's RankingMetrics class.**
    - **pass in an RDD of key-value pairs to our own average precision function**
      - **the key is an Array of predicted item IDs for a user**
      - **the value is an array of actual item IDs**
    - **Calculate MAP using RankingMetrics**

```
➢ import org.apache.spark.mllib.evaluation.RankingMetrics
➢ val predictedAndTrueForRanking = allRecs.join(userMovies).map{
      case (userId,    (predicted, actualWithIds)) =>
      val actual = actualWithIds.map(_._2)
      (predicted.toArray, actual.toArray)}
➢ val rankingMetrics = new RankingMetrics(predictedAndTrueForRanking)
➢ println("Mean Average Precision = " + rankingMetrics.meanAveragePrecision)
```

```
scala> println("Mean Average Precision = " + rankingMetrics.meanAveragePrecision)
Mean Average Precision = 0.18155054038225935
```

Evaluating the performance of models

- **Using Mllib's built-in evaluation functions**
  - **MAP**
    - **Compute MAP**

```
➢val MAPK2000 = allRecs.join(userMovies).map{
    case (userId, (predicted, actualWithIds)) =>
    val actual = actualWithIds.map(_._2).toSeq
    avgPrecisionK(actual, predicted, 2000)
}.reduce(_ + _) / allRecs.count
➢println("Mean Average Precision = " + MAPK2000)
```

```
scala> println("Mean Average Precision = " + MAPK2000)
Mean Average Precision = 0.18155054038222595
```

    - **The MAP from our own function is the same as the own computed using RankingMetrics**

Summary

- **Use Spark's MIlib library to train a collaborative filtering recommendation model**
  - **How to use the model to make predictions for the items**
  - **Use the model to find items that are similar or related to a given item**
  - **Explore common metrics to evaluate the predictive capability of our recommendation model**

Thank you