CSE 424
Big Data

Spark MLlib

Slides 8

Instructor: Asst. Prof. Dr. Hüseyin ABACI

Outline

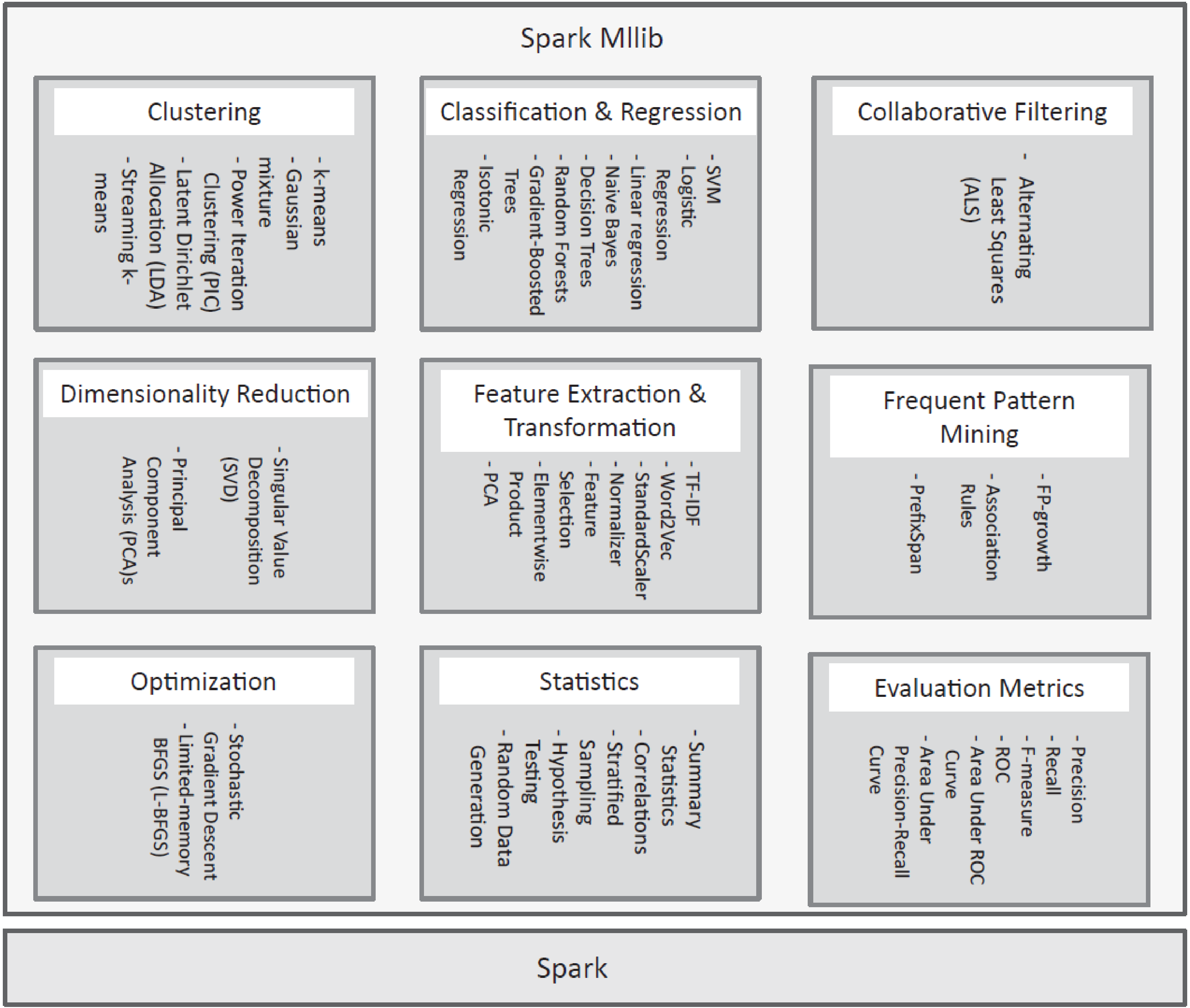
- Spark MLlib
- Movie Recommendation Systems - Movie Dataset
- ALS for MovieLens 100K
- MovieLens 100K – Recommending Users
- MovieLens 100K – Inspecting Recommendations
- MovieLens 100K – Inspecting Recommendations
- MovieLens 100K – Performance Evaluation

Bahga, Arshdeep, and Vijay Madisetti. Big data science & analytics: A hands-on approach. VPT, 2016.
Pentreath, Nick. *Machine learning with spark*. Packt Publishing Ltd, 2015.

Spark MLlib

- Spark MLlib is the Spark's machine learning library which provides implementations of various machine learning algorithms including classification, regression, clustering, collaborative filtering and dimensionality reduction.
- The MLlib APIs are built on top of the Spark's resilient distributed datasets (RDDs). MLlib also provides high-level data types such as Vector, LabeledPoint, Rating and Matrix, which are backed by RDDs.
- The benefit of using MLlib over machine learning libraries is that it provides parallel implementations of machine learning algorithms and can process large distributed datasets.
- Spark Mllib provides APIs for Python, Scala, and Java programming languages.

•



Spark Mllib
Clustering
- k-means
- Gaussian mixture
- Power Iteration Clustering (PIC)
- Latent Dirichlet Allocation (LDA)
- Streaming k-means
Classification & Regression
- SVM
- Logistic Regression
- Linear regression
- Naive Bayes
- Decision Trees
- Random Forests
- Gradient-Boosted Trees
- Isotonic Regression
Collaborative Filtering
- Alternating Least Squares (ALS)
Dimensionality Reduction
- Singular Value Decomposition (SVD)
- Principal Component Analysis (PCA)s
Feature Extraction & Transformation
- TF-IDF
- Word2Vec
- StandardScaler
- Normalizer
- Feature Selection
- Elementwise Product
- PCA
Frequent Pattern Mining
- FP-growth
- Association Rules
- PrefixSpan
Optimization
- Stochastic Gradient Descent
- Limited-memory BFGS (L-BFGS)
Statistics
- Summary Statistics
- Correlations
- Stratified Sampling
- Hypothesis Testing
- Random Data Generation
Evaluation Metrics
- Precision
- Recall
- F-measure
- ROC
- Area Under ROC Curve
- Area Under Precision-Recall Curve
Spark

Movie Recommendation Systems

- Let us now look at an example of a system for making movie recommendations using the collaborative filtering approach.
- First, we will explore the data sets and visualise the data then Python implementation of the recommendation system that uses the Spark MLlib's implementation of the Alternating Least Squares (ALS) algorithm.
- For this example, we will use the MovieLens dataset * which includes ratings given by users to movies. For development purpose, a smaller version of the dataset (MovieLens 100K) which includes 100,000 ratings from 943 users on 1682 movies, is used. For testing the working code with a big dataset, you can use the MovieLens 20M dataset which includes 20 million ratings applied to 27,000 movies by 138,000 users.
- Data sets consist of three text files namely: User dataset "u.user" that contains the information about users (age, occupation etc.), Rating dataset "u.data" ratings given by which user (user id, rating, timestamp etc.) and Movie dataset "u.item" that contains information about the each movie (name of the movie, year, url etc.) .

* http://files.grouplens.org/datasets/movielens/ml-100k.zip

Movie Recommendation Systems - Movie Dataset

- The u.item file contains the movie id, title, release data, and IMDB link fields and a set of fields related to movie category data. It is also separated by a | character:

```
movie_data = sc.textFile('./ml-100k/u.item')
movie_data.first()
```

```
'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'
```

```
num_movies = movie_data.count()
print('Number of Movies', num_movies)
```

```
Number of Movies 1682
```

Movie Recommendation Systems - Movie Dataset

- In the following code block, we can see that we need a small function called *convert_year* to handle errors (replacing with "1900") in the parsing of the release date field. This is due to some bad data in one line of the movie data:

```
def convert_year(x):
 try:
     return int(x[-4:])
 except:
     return 1900
```

- Once we have our utility function to parse the year of release, we can apply it to the movie data using a map transformation and collect the results:

```
movie_fields = movie_data.map(lambda lines: lines.split('|'))
years = movie_fields.map(lambda fields: fields[2]).map(lambda x: convert_year(x))
```

- Since we have assigned the value 1900 to any error in parsing, we can filter these bad values out of the resulting data using Spark's filter transformation:

```
years_filtered = years.filter(lambda x: x != 1900)
```

Movie Recommendation Systems - Movie Dataset

```
movie_fields.first()
```

```
['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']
```

```
years.take(10)
```

```
[1995, 1995, 1995, 1995, 1995, 1995, 1995, 1995, 1995, 1996]
```

Movie Recommendation Systems - Movie Dataset

- After filtering out bad data, we will transform the list of movie release years into movie ages by subtracting the current year, use *countByValue* to compute the counts for each movie age, and finally, plot our histogram of movie ages (again, using the hist function, where the values variable are the values of the result from countByValue, and the bins variable are the keys):

```
movie_ages = years_filtered.map(lambda yr: 1998-yr).countByValue()
values = list(movie_ages.values())
bins = list(movie_ages.keys())
```

Movie Recommendation Systems - Movie Dataset

- In order to use matplotlib and array operations we need to import followings:

```
import matplotlib.pyplot as plt
```

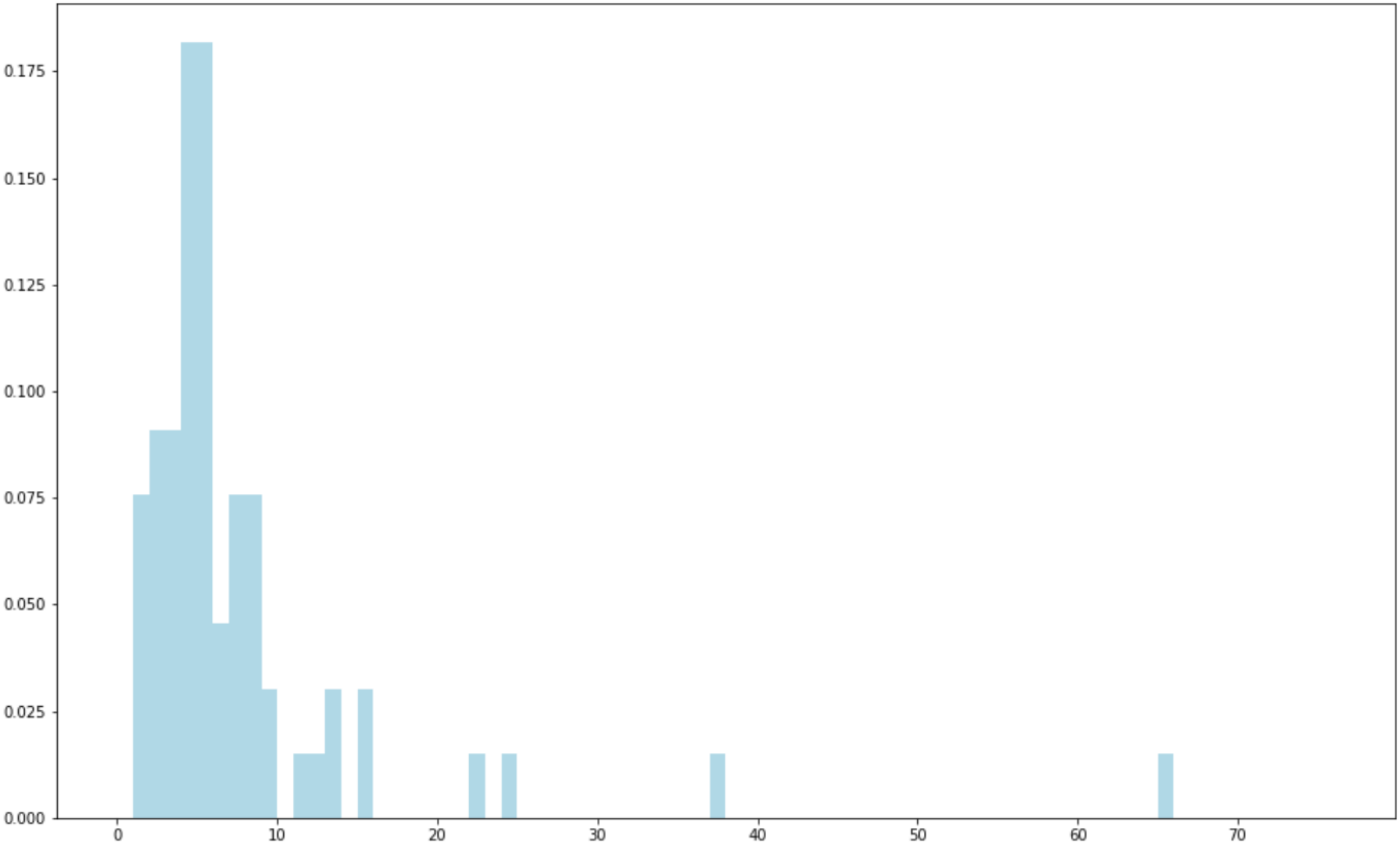
```
import numpy as np
```

```
# matplotlib "bins" accepts only sorted lists
bins = np.sort(bins)
print(bins)
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 72 76]
```

```
plt.hist(values, bins=bins, color='lightblue', density=True)
fig = plt.gcf()
fig.set_size_inches(16,10)
```

Movie Recommendation Systems - Movie Dataset



Movie Recommendation Systems - User Dataset

- First, we will analyze the characteristics of MovieLens users (user id, age, gender, occupation, and ZIP code fields).

```
In [3]: user_data = sc.textFile('./ml-100k/u.user')
        user_data.first()

Out[3]: '1|24|M|technician|85711'
```

- Let's transform the data by splitting each line, around the "|" character. This will give us an RDD where each record is a Python list that contains the user ID, age, gender, occupation, and ZIP code fields.
- We will then count the number of users, genders, occupations, and ZIP codes. We can achieve this by running the following code in the console, line by line.

```
user_fields = user_data.map(lambda line: line.split('|'))
num_users = user_fields.map(lambda field: field[0]).count()
num_genders = user_fields.map(lambda field: field[2]).distinct().count()
num_occupations = user_fields.map(lambda field: field[3]).distinct().count()
num_zipcodes = user_fields.map(lambda field: field[4]).distinct().count()
print('Number of User: {}, Gender: {}, Occupation: {}, ZipCode: {}'.format(num_users, num_genders, num
_occupations, num_zipcodes))
```

```
Number of User: 943, Gender: 2, Occupation: 21, ZipCode: 795
```

Movie Recommendation Systems - User Dataset

- Next, we will create a histogram to analyze the distribution of user ages, using matplotlib's hist function.
- Before that, we will calculate number of users for each age group.

```
num_ages = user_fields.map(lambda field: (field[1], 1)).reduceByKey(lambda x, y: x + y)
for i in num_ages.collect():
    print(i)
```

```
('24', 33)
('53', 12)
('33', 26)
('57', 9)
('29', 32)
('45', 15)
('21', 27)
```

Movie Recommendation Systems - User Dataset

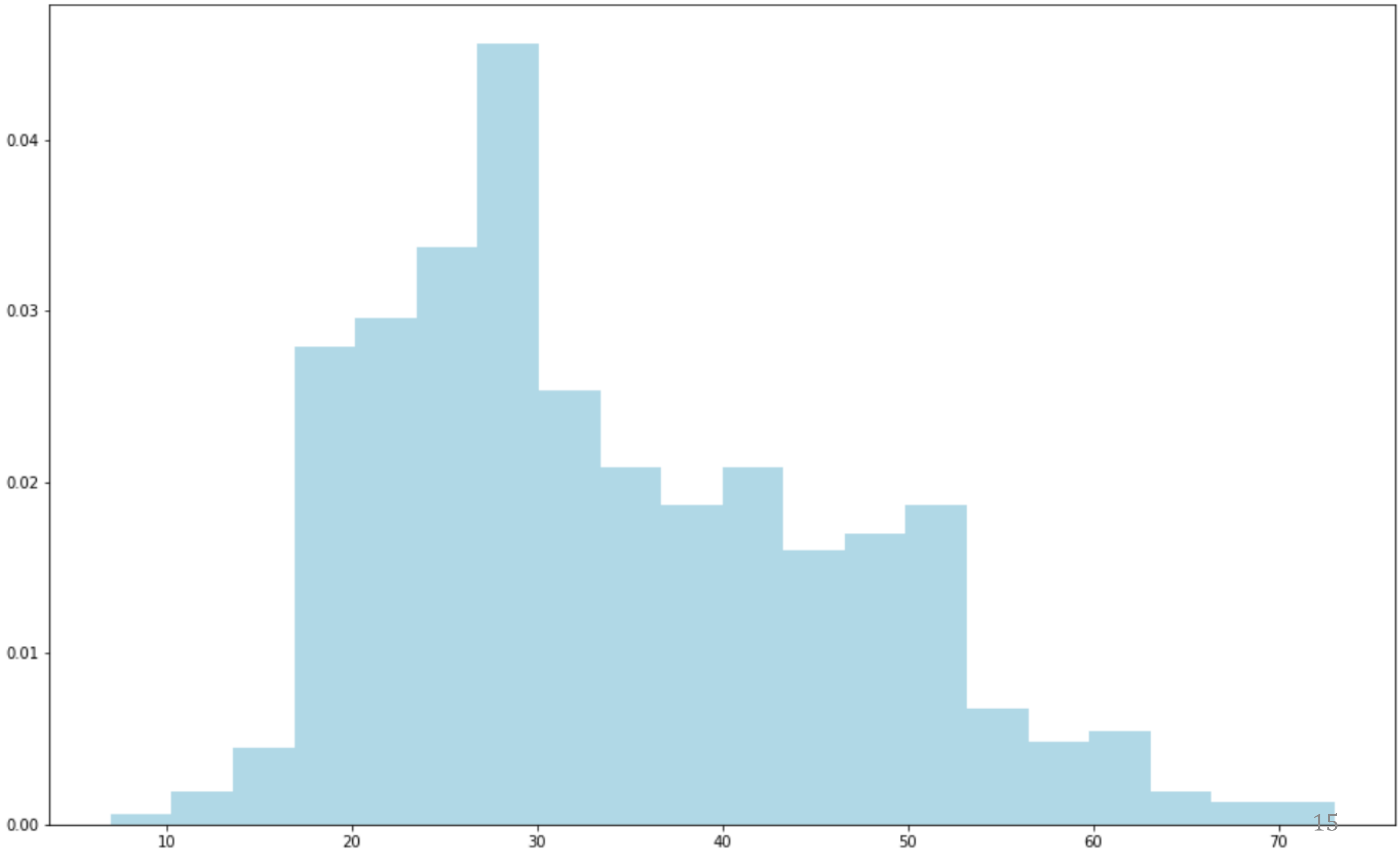
- We will pass in the num_ages array, together with the number of bins for our histogram (20 in this case), to the hist function. Using the density=True argument, we also specified that we want the histogram to be normalized so that each bucket represents the percentage of the overall data that falls into that bucket.

```
ages = user_fields.map(lambda x: int(x[1])).collect()
plt.hist(ages, bins=20, color='lightblue', density=True)
fig = plt.gcf()
fig.set_size_inches(16, 10)
```

- If your ages are distributed between 1-100, with bins=20 you will get 100/20 = 5 ages in same bin. Bins are grouped 5 by 5 in above example. For example number of users in age 10, 11, 12, 13, 14 summed into one bin.

Movie Recommendation Systems - User Dataset

- As we can see, the ages of MovieLens users are somewhat skewed towards younger viewers. A large number of users are between the ages of about 15 and 35.



Movie Recommendation Systems - Rating Dataset

- Let's now take a look at the ratings data (user id, movie id, rating (1-5 scale), and timestamp fields). There are 100,000 ratings, and unlike the user and movie datasets, these records are split with a tab character ("\t").

```
rating_data = sc.textFile('./ml-100k/u.data')
rating_data.take(10)
```

```
['196\t242\t3\t881250949',
 '186\t302\t3\t891717742',
 '22\t377\t1\t878887116',
 '244\t51\t2\t880606923',
 '166\t346\t1\t886397596',
 '298\t474\t4\t884182806',
 '115\t265\t2\t881171488',
 '253\t465\t5\t891628467',
 '305\t451\t3\t886324817',
 '6\t86\t3\t883603013']
```

```
num_ratings = rating_data.count()
print('Number of Rating: ', num_ratings)
```

```
Number of Rating:  100000
```

Movie Recommendation Systems - Rating Dataset

- In order to compute some basic summary statistics and frequency histograms for the rating values.
- Spark provides a stats function for RDDs; this function contains a numeric variable (such as ratings in this case) to compute similar summary statistics:

```
rating_fields = rating_data.map(lambda line: line.split('\t'))
ratings = rating_fields.map(lambda field: int(field[2]))
ratings.stats()
```

```
(count: 100000, mean: 3.5298600000000024, stdev: 1.12566797076251, max: 5.0, min: 1.0)
```

- Looking at the results, the average rating given by a user to a movie is around 3.5, so we might expect that the distribution of ratings will be skewed towards slightly higher ratings. Let's see whether this is true by creating a bar chart of rating values using a similar procedure as we did for occupations:

Movie Recommendation Systems - Rating Dataset

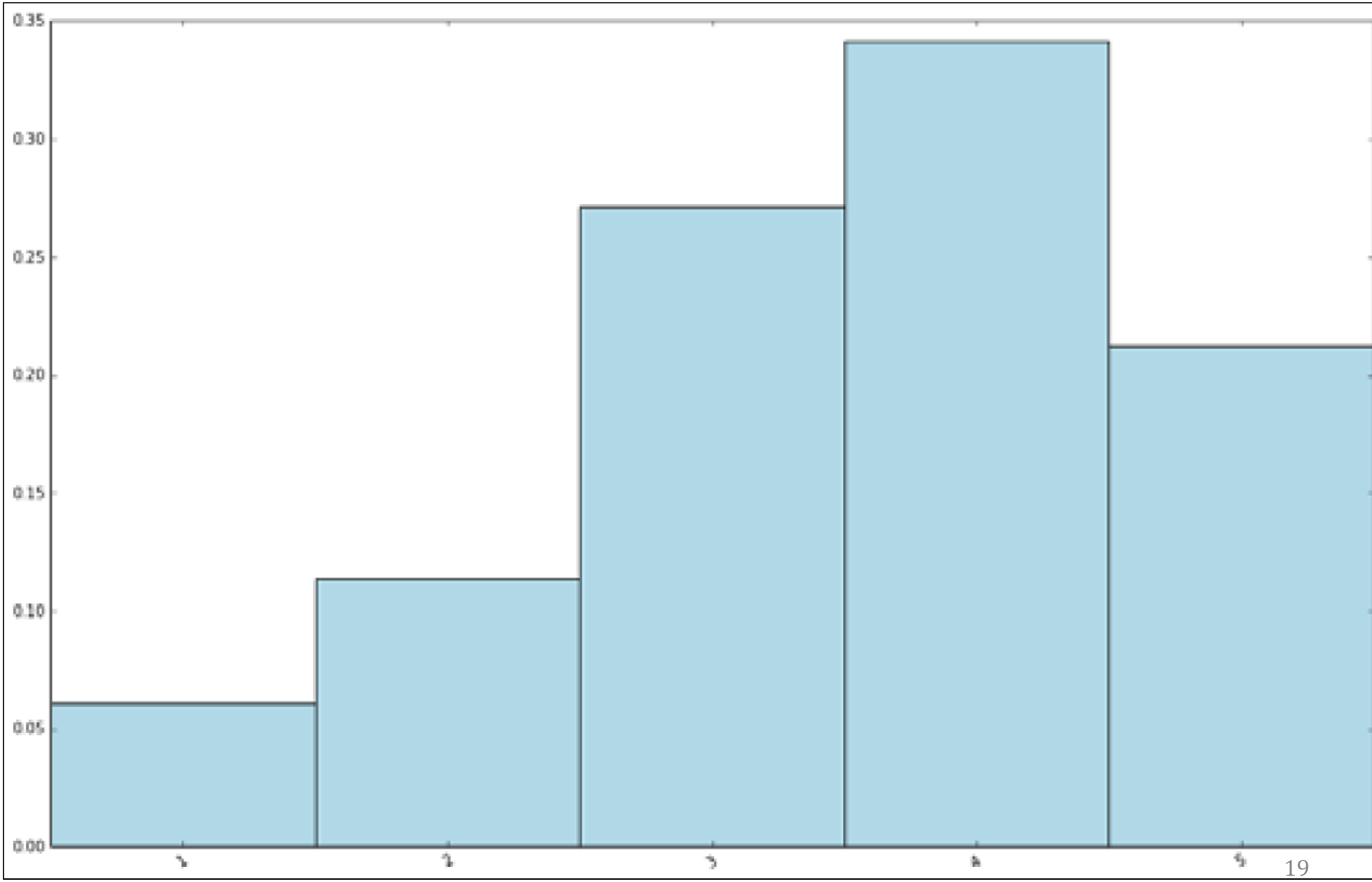
```
import numpy as np
count_by_rating = ratings.countByValue()
x_axis = np.array(count_by_rating.keys())
y_axis = np.array([float(c) for c in count_by_rating.values()])

# we normalize the y-axis here to percentages
y_axis_normed = y_axis / y_axis.sum()
pos = np.arange(len(x_axis))
width = 1.0

ax = plt.axes()
ax.set_xticks(pos + (width / 2))
ax.set_xticklabels(x_axis)

plt.bar(pos, y_axis_normed, width, color='lightblue')
plt.xticks(rotation=30)
fig = plt.gcf()
fig.set_size_inches(16, 10)
```

Movie Recommendation Systems - Rating Dataset



Recommendation Engine – MovieLens 100K

```
raw_ratings = rating_data.map(lambda x: x.split("\t")[:3])
raw_ratings.take(5)
```

```
[['196', '242', '3'],
 ['186', '302', '3'],
 ['22', '377', '1'],
 ['244', '51', '2'],
 ['166', '346', '1']]
```

Recommendation Engine – MovieLens 100K

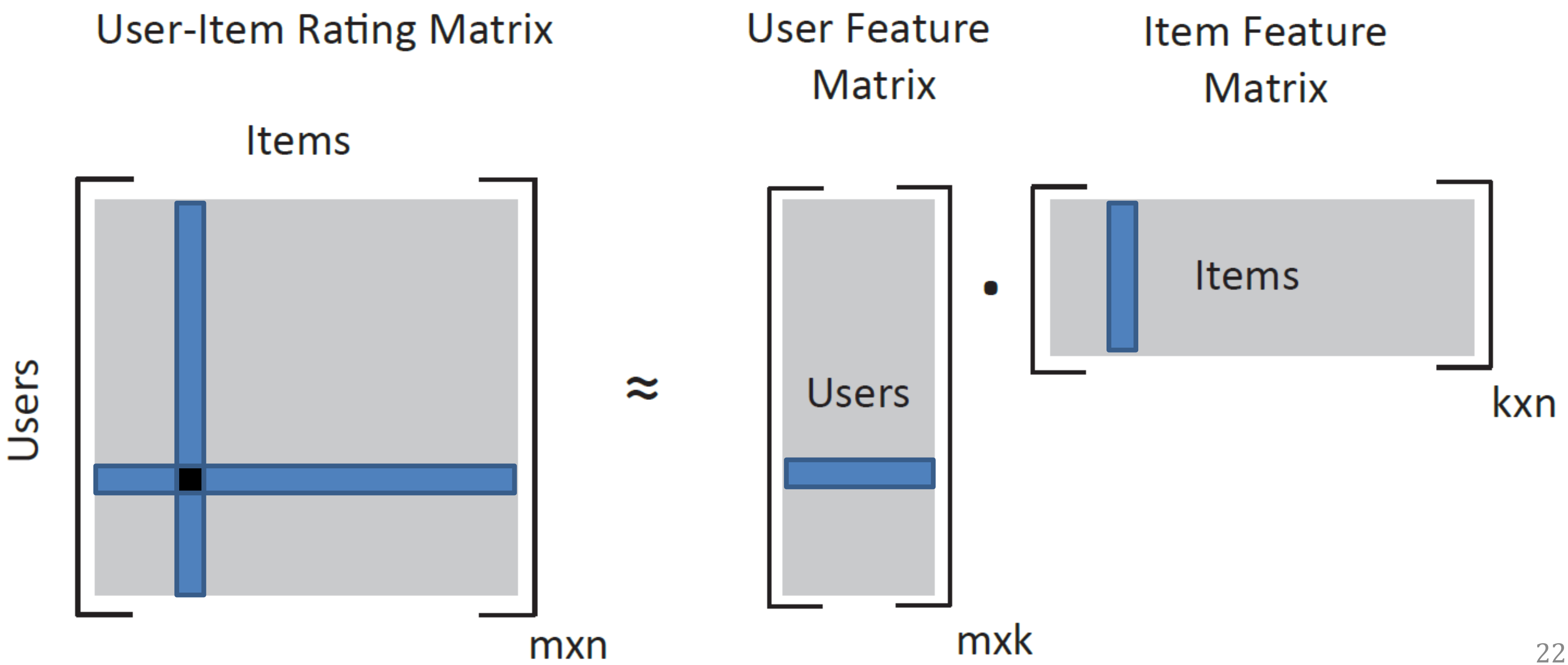
```
from pyspark.mllib.recommendation import ALS # Alternating Least Squares
from pyspark.mllib.recommendation import Rating

ratings = raw_ratings.map(lambda x: Rating(int(x[0]), int(x[1]), float(x[2])))
ratings.take(10)
```

```
[Rating(user=196, product=242, rating=3.0),
 Rating(user=186, product=302, rating=3.0),
 Rating(user=22, product=377, rating=1.0),
 Rating(user=244, product=51, rating=2.0),
 Rating(user=166, product=346, rating=1.0),
 Rating(user=298, product=474, rating=4.0),
 Rating(user=115, product=265, rating=2.0),
 Rating(user=253, product=465, rating=5.0),
 Rating(user=305, product=451, rating=3.0),
 Rating(user=6, product=86, rating=3.0)]
```

Alternating Least Squares (ALS)- We've seen this before !

- In this section, we will describe a model-based collaborative filtering approach based on Alternating Least Squares (ALS) algorithm.
- ALS works by iteratively solving a series of least squares regression problems. In each iteration, one of the user- or 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).



Alternating Least Squares (ALS)- We've seen this before !

- Let us formulate the collaborative filtering problem. Let

m = number of users
n= number of items
k = number of latent factors (or number of user/item features)
$r^{(u,i)}$ = rating given by user u to item i
$w(i, j)$ = 1 if user i has rated item j and 0 otherwise
$x^{(u)}$ = feature vector for user u
$y^{(i)}$ = feature vector for item i

- Figure shows (in previous slide) a user rating matrix where each row belongs to a user and the columns are the ratings given to items. Given the user-item rating matrix, the learning objective is to learn the user and item latent features (that represent the user preferences and item features).
- In other words, given an *m x n* dimensional user-item matrix, we want to factorize the matrix into an *m x k* matrix (user feature vector) and *k x n* matrix (item feature vector).

Alternating Least Squares (ALS)

- We're now ready to train our model! The other inputs required for our model are as follows:
  - **Rank (latent factor):** This refers to the number of factors in our ALS model, that is, the number of hidden features in our low-rank approximation matrices. Generally, the greater the number of factors, the better, but this has a direct impact on memory usage, both for computation and to store models for serving, particularly for large number of users or items. Hence, this is often a trade-off in real-world use cases. A rank in the range of 10 to 200 is usually reasonable.
  - **iterations:** This refers to the number of iterations to run. While each iteration in ALS is guaranteed to decrease the reconstruction error of the ratings matrix, ALS models will converge to a reasonably good solution after relatively few iterations. So, we don't need to run for too many iterations in most cases (around 10 is often a good default).
  - **lambda:** This parameter controls the regularization of our model. Thus, lambda controls over fitting. The higher the value of lambda, the more is the regularization applied. What constitutes a sensible value is very dependent on the size, nature, and sparsity of the underlying data, and as with almost all machine learning models, the regularization parameter is something that should be tuned using out-of-sample test data and cross-validation approaches.

Alternating Least Squares (ALS)

Parameters:

ratings – RDD of *Rating* or (userID, productID, rating) tuple.

rank – Rank of the feature matrices computed (number of features).

iterations – Number of iterations of ALS. (default: 5)

lambda – Regularization parameter. (default: 0.01)

blocks – Number of blocks used to parallelize the computation. A value of -1 will use an auto-configured number of blocks. (default: -1)

nonnegative – A value of True will solve least-squares with nonnegativity constraints. (default: False)

seed – Random seed for initial matrix factorization model. A value of None will use system time as the seed. (default: None)

Alternating Least Squares (ALS)

- We'll use rank of 50, 10 iterations, and a lambda parameter of 0.01 to illustrate how to train our model:

```
model = ALS.train(ratings, 50, 10, 0.01)
```

- This returns a *MatrixFactorizationModel* object, which contains the user and item factors in the form of an RDD of (id, factor) pairs. These are called *userFeatures* and *productFeatures*, respectively. For example:

```
model.userFeatures()
```

```
PythonRDD[312] at RDD at PythonRDD.scala:53
```

```
model.userFeatures().count()
```

```
943
```

Alternating Least Squares (ALS)

```
item_id = 567
item_vector = model.productFeatures().lookup(item_id)[0]
item_vector
```

```
array('d', [0.4837658703327179, -0.1846004426479339б, -0.3169634938240051, -0.6
529982686042786, -0.2664172053337097, -0.1978134214878082З, -0.448584496974945О
7, -0.4342204332351684б, 0.528467357158660Ə, 1.0446935892105103, 0.72407245635Ə
8633, 0.3871027529239654б, -0.482578486204147З4, 0.0902467221021652Ʃ, 0.4597048
1634140015, -0.5229714512825012, -0.6255507469177246, -0.3891392350196838Ɐ, 0.3
13943505287170Ⅎ, -0.08101335912942886, -0.3322042226791382, -0.0499013960361480
```

Alternating Least Squares (ALS)

- The MatrixFactorizationModel class has a convenient predict method that will compute a predicted score for a given user and item combination:

```
predicted_rating = model.predict(789, 123)
print(predicted_rating)
# You might see different result in every train because the ALS model is initialized randomly
```

```
4.301980734473551
```

- We can use this method to make predictions for many users and items at the same time.
- To generate the *top-K* recommended items for a user and user for items MatrixFactorizationModel provides a convenience methods called *recommendProducts and recommendUsers*. This takes two arguments: userID/itemID and num, where num is the number of items to recommend.
- It returns the top num items ranked in the order of the predicted score. Here, the scores are computed as the dot product between the user-factor vector and each item-factor vector.
- Let's generate the top 10 recommended items for user 789:

Alternating Least Squares (ALS)

```
userId = 789
K = 10
top_k_recs = model.recommendProducts(userId, K)
for i in top_k_recs:
    print(i)
```

```
Rating(user=789, product=195, rating=5.663019138696582)
Rating(user=789, product=177, rating=5.523128151316249)
Rating(user=789, product=429, rating=5.452148109847103)
Rating(user=789, product=302, rating=5.302996495616345)
Rating(user=789, product=182, rating=5.2548182500276726)
Rating(user=789, product=447, rating=5.18594830266391)
Rating(user=789, product=32, rating=5.175750754734843)
Rating(user=789, product=603, rating=5.144425958361632)
Rating(user=789, product=101, rating=5.075492891819342)
Rating(user=789, product=276, rating=5.01041935547627)
```

MovieLens 100K – Recommending Users

- Recommend the top-K users for a given product

```
productID =465
K = 5

topKitem = model.recommendUsers(productID, K)

for i in topKitem:
      print(i)
```

```
#Output:  [Rating(user=519, product=465, rating=7.504947875474900 2),
#Rating(user=180, product=465, rating=7.347811316007009 1),
#Rating(user=217, product=465, rating=7.219420195217776 6),
#Rating(user=808, product=465, rating=6.539883949632426 6),
#Rating(user=93, product=465, rating=6.498897177019603 8)]
```

MovieLens 100K – Inspecting Recommendations

```
movie_fields.first()
```

```
['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']
```

- We can give these recommendations a sense check by taking a quick look at the titles of the movies a user has rated and the recommended movies next slides shows this.

MovieLens 100K – Inspecting Recommendations

```
titles
```

```
{1: 'Toy Story (1995)',
 2: 'GoldenEye (1995)',
 3: 'Four Rooms (1995)',
 4: 'Get Shorty (1995)',
 5: 'Copycat (1995)',
 6: 'Shanghai Triad (Yao a yao yao
 7: 'Twelve Monkeys (1995)',
 8: 'Babe (1995)',
 9: 'Dead Man Walking (1995)',
 10: 'Richard III (1995)'
```

```
titles[1]
```

```
'Toy Story (1995)'
```

MovieLens 100K – Inspecting Recommendations

- collectAsMap() : Return the key-value pairs in this RDD to the master as a dictionary.

```
>>> m = sc.parallelize([(1, 2), (3, 4)]).collectAsMap()
>>> m[1]
2
>>> m[3]
4
```

```
dict0 = {
     'fname':'Jeff',
     'lname':'Aven',
     'pos':'author'
}
```

- We'll map the required fields first and second map convert this data to a key-value pairs (mapping the movie ID to the title), then collectAsMap() converts key–value pairs to a dictionary:

```
titles = movie_fields.map(lambda line: line[:2])
.map(lambda x: (int(x[0]), x[1]))
.collectAsMap()
```

MovieLens 100K – Inspecting Recommendations

- For our user 789, we can find out what movies they have rated. We will do this now by first using the *keyBy* Spark function to create an RDD of key-value pairs from our *ratings* RDD, where the key will be the user ID. We will then use the *lookup* function to return just the ratings for this key (that is, that particular user ID) to the driver:

```
movies_for_user = ratings.keyBy(lambda x: x.user)
```

```
movies_for_user.take(10)
```

```
[(196, Rating(user=196, product=242, rating=3.0)),
 (186, Rating(user=186, product=302, rating=3.0)),
 (22, Rating(user=22, product=377, rating=1.0)),
 (244, Rating(user=244, product=51, rating=2.0)),
 (166, Rating(user=166, product=346, rating=1.0)),
 (298, Rating(user=298, product=474, rating=4.0)),
 (115, Rating(user=115, product=265, rating=2.0)),
 (253, Rating(user=253, product=465, rating=5.0)),
 (305, Rating(user=305, product=451, rating=3.0)),
 (6, Rating(user=6, product=86, rating=3.0))]
```

MovieLens 100K – Inspecting Recommendations

```
movies_for_user = ratings.keyBy(lambda x: x.user).lookup(789)
movies_for_user
```

```
[Rating(user=789, product=1012, rating=4.0),
 Rating(user=789, product=127, rating=5.0),
 Rating(user=789, product=475, rating=5.0),
 Rating(user=789, product=93, rating=4.0),
 Rating(user=789, product=1161, rating=3.0),
 Rating(user=789, product=286, rating=1.0),
 Rating(user=789, product=293, rating=4.0),
 Rating(user=789, product=9, rating=5.0),
 Rating(user=789, product=50, rating=5.0),
 Rating(user=789, product=294, rating=3.0),
 Rating(user=789, product=181, rating=4.0)
```

```
len(movies_for_user)
```

```
33
```

MovieLens 100K – Inspecting Recommendations

- Next, we will take the 10 movies with the highest ratings by sorting the *movies_for_user* collection using the *rating* field of the *Rating* object.
- We will then extract the movie title for the relevant product ID attached to the *Rating* class (and dictionary created before) from our mapping of movie titles and print out the top 10 titles with their ratings:

```
movies_for_user.sort(reverse = True, key = lambda x: x.rating)

sc.parallelize(movies_for_user[:10])
.map(lambda rating: (titles[rating.product], rating.rating))
.collect()
```

```
[('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)]
```

MovieLens 100K – Inspecting Recommendations

```python
userId = 789
K = 10
top_k_recs = model.recommendProducts(userId, K)
for i in top_k_recs:
    print(i)
```

```
Rating(user=789, product=607, rating=5.74494665558465)
Rating(user=789, product=214, rating=5.506148925675795)
Rating(user=789, product=47, rating=5.39184497259312)
Rating(user=789, product=56, rating=5.35547301620482)
Rating(user=789, product=179, rating=5.33858625366794)
Rating(user=789, product=192, rating=5.3343956073021)
Rating(user=789, product=466, rating=5.3104068425896385)
Rating(user=789, product=46, rating=5.234452865710505)
Rating(user=789, product=176, rating=5.2251634871861015)
Rating(user=789, product=772, rating=5.101071332842363)
```

MovieLens 100K – Inspecting Recommendations

- Now, let's take a look at the top 10 recommendations for this user (789) and see what the titles are using the same approach as the one we used earlier (note that the recommendations are already sorted):

```
sc.parallelize(top_k_recs)
.map(lambda rating: (titles[rating.product], rating.rating))
.collect()
```

```
[('Terminator, The (1984)', 5.663019138696582),
 ('Good, The Bad and The Ugly, The (1966)', 5.523128151316249),
 ('Day the Earth Stood Still, The (1951)', 5.452148109847103),
 ('L.A. Confidential (1997)', 5.302996495616345),
 ('GoodFellas (1990)', 5.254818250027672 6),
 ('Carrie (1976)', 5.18594830266391),
 ('Crumb (1994)', 5.17575075473 4843),
 ('Rear Window (1954)', 5.144425958361632),
 ('Heavy Metal (1981)', 5.075492891819342),
 ('Leaving Las Vegas (1995)', 5.0104193554 7627)]
```

MovieLens 100K – Inspecting Recommendations

```
[('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)]
```

User rated

ALS recommend.

```
[('Terminator, The (1984)', 5.663019138696582),
 ('Good, The Bad and The Ugly, The (1966)', 5.523128151316249),
 ('Day the Earth Stood Still, The (1951)', 5.452148109847103),
 ('L.A. Confidential (1997)', 5.302996495616345),
 ('GoodFellas (1990)', 5.2548182500276726),
 ('Carrie (1976)', 5.18594830266391),
 ('Crumb (1994)', 5.175750754734843),
 ('Rear Window (1954)', 5.144425958361632),
 ('Heavy Metal (1981)', 5.075492891819342),
 ('Leaving Las Vegas (1995)', 5.01041935547627)]
```

MovieLens 100K – Item Recommendations

We've seen this before !

- Item recommendations are about answering the following question: for a certain item, what are the items most similar to it? Here, 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 to produce a single value.
- Common similarity measures include
  - Euclidean Distance
  - Manhattan Distance
  - Minkowski Distance
  - Jaccard Similarity
  - Cosine Similarity

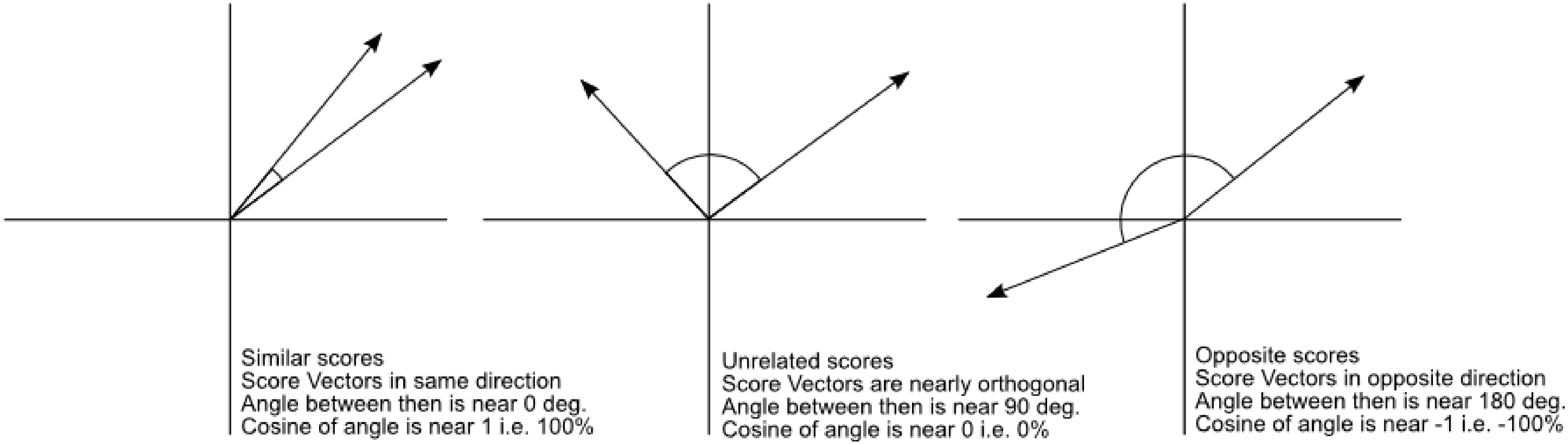
MovieLens 100K – Item Recommendations

We've seen this before !

- Cosine similarity is a measure of the angle between two vectors in an n-dimensional space. It is computed by first calculating the dot product between the vectors and then dividing the result by a denominator, which is the norm (or length) of each vector multiplied together (specifically, the L2-norm is used in cosine similarity). In this way, cosine similarity is a normalized dot product.
- The cosine similarity measure takes on values between -1 and 1. A value of 1 implies completely similar, while a value of 0 implies independence (that is, no similarity).
- This measure is useful because it also captures negative similarity, that is, a value of -1 implies that not only are the vectors not similar, but they are also completely opposite (dissimilar).

MovieLens 100K – Item Recommendations

We've seen this before !



$$\text{similarity} = \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\| \|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \sqrt{\sum_{i=1}^{n} B_i^2}}$$

MovieLens 100K – Item Recommendations Cosine Similarities

- 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.
- We will use the cosine similarity metric, and we will use the numpy linear algebra library to compute the required vector dot products.

- This is similar to how the existing predict and *recommendProducts* methods work, except that we will use cosine similarity as opposed to just the dot product.
- We will need a method to compute the cosine similarity between two vectors. Let's create our *cosineSimilarity* function here:

```
def cosineSimilarity(item_id, a, b):
    dot = np.dot(a, b)
    norma = np.linalg.norm(a)
    normb = np.linalg.norm(b)
    cos = dot / (norma * normb)
    return item_id, cos
```

MovieLens 100K – Item Recommendations

- Let's try it out on one of our item factors for item 567. We will need to collect an item factor from our model; we will do this using the lookup method in a similar way that we did earlier to collect the ratings for a specific user.
- In the following lines of code, we also use the head function, since lookup returns an array of values, and we only need the first value (in fact, there will only be one value, which is the factor vector for this item).

```
item_id = 567
item_vector = model.productFeatures().lookup(item_id)[0]
item_vector
```

```
array('d', [0.4837658703327179, -0.18460044264793396, -0.3169634938240051, -0.6
529982686042786, -0.2664172053337097, -0.1978134214878082З, -0.448584496974945O
7, -0.4342204323516846, 0.528467357158609, 1.044693589210510З, 0.72407245635 9
8633, 0.3871027529239654 5, -0.4825784862041473 4, 0.0902467221021652 2, 0.4597048
1634140015, -0.522971451282501 2, -0.625550746917724 6, -0.389139235019683 84, 0.3
139435052871704, -0.08101335912942886, -0.3322042226791382, -0.0499013960361480
```

MovieLens 100K – Item Recommendations

```
print(titles[item_id])
```

```
Wes Craven's New Nightmare (1994)
```

```
cosineSimilarity(item_id, item_vector, item_vector)
```

```
(567, 0.9999999999999998)
```

MovieLens 100K – Item Recommendations

- A similarity metric should measure how close, in some sense, two vectors are to each other. Here, we can see that our cosine similarity metric tells us that this item vector is identical to itself, which is what we would expect:

```
sims = model.productFeatures().map(lambda data: cosineSimilarity(data[0], data
[1], item_vector))
```

```
sims.top(10, key=lambda x: x[1])
```

```
[(567, 0.99999999999999998),
 (563, 0.75800073878329659),
 (1244, 0.69645799739870238),
 (665, 0.69126653824594353),
 (201, 0.68939838748026124),
 (670, 0.68456962026496087),
 (1007, 0.68110076811230572),
 (184, 0.67842455840156993),
 (940, 0.67815052516745540),
 (1012, 0.67659236741208323)]
```

MovieLens 100K – Performance Evaluation

- The Mean Squared Error (MSE) is a direct measure of the reconstruction error of the user-item rating matrix. It is also the objective function being minimized in certain models, specifically many matrix-factorization techniques, including ALS.
- It is defined as the sum of the squared errors divided by the number of observations. The squared error, in turn, is the square of the difference between the predicted rating for a given user-item pair and the actual rating.

```
data = sc.textFile("file:///home/hadoop/ml-100k/u.data")

(trainingRatings, testRatings) = data.randomSplit([0.7, 0.3])
```

MovieLens 100K – Performance Evaluation

```
trainingRatings.take(10)
```

```
['186\t302\t3\t891717742',
 '22\t377\t1\t878887116',
 '244\t51\t2\t880606923',
 '298\t474\t4\t884182806',
 '115\t265\t2\t881171488',
 '305\t451\t3\t886324817',
 '200\t222\t5\t876042340',
 '210\t40\t3\t891035994',
 '224\t29\t3\t888104457',
 '303\t785\t3\t879485318']
```

```
testRatings.take(10)
```

```
['196\t242\t3\t881250949',
 '166\t346\t1\t886397596',
 '253\t465\t5\t891628467',
 '6\t86\t3\t883603013',
 '62\t257\t2\t879372434',
 '286\t1014\t5\t879781125',
 '234\t1184\t2\t892079237',
 '291\t118\t2\t874833878',
 '50\t246\t3\t877052329',
 '276\t796\t1\t874791932']
```

MovieLens 100K – Performance Evaluation

```
trainingData = trainingRatings.map(lambda l: l.split('\t'))
.map(lambda l: Rating(int(l[0]), int(l[1]), float(l[2])))

trainingData.first()
#Output: Rating(user=196, product=242, rating=3.0)

testData = testRatings.map(lambda l: l.split('\t'))
.map(lambda l: (int(l[0]), int(l[1])))

testData.first()
#Output: (244, 51)

# Build the recommendation model using Alternating Least
Squares
rank = 10
numIterations = 50
model = ALS.train(trainingData, rank, numIterations)
```

MovieLens 100K – Performance Evaluation

```
#Predict rating for the given user and product.
model.predict(253, 465)
#Output: 4.5738394508197189

#Return a list of predicted ratings for input user and product
#pairs
predictions = model.predictAll(testData)
```

```
predictions.first()
```

```
Rating(user=452, product=384, rating=2.0326541596754826)
```

MovieLens 100K – Performance Evaluation

- In order to calculate MSE we may create two np arrays, because easier to operate .
- Here we create array for predictions.

```
allPredictedRatings=predictions.map(lambda l: (float(l[2])))
```

```
allPredictedRatings.take(5)
```

```
[2.032654159675482б,
 2.719842253447941,
 2.247847303623934,
 2.966962645958879,
 3.0392122216130835]
```

```
arr_allPredictedRatings=allPredictedRatings.collect()#returns an array
```

MovieLens 100K – Performance Evaluation

- Here we create array for original rates.

```
testData = testRatings.map(lambda l: l.split('\t')).map(lambda l: (int(l[0]), int(l[1])))
```

```
allTestRatings=testRatings.map(lambda l: l.split('\t')).map(lambda l: (float(l[2])))
```

```
allTestRatings.take(5)
```

```
[3, 1, 5, 3, 2]
```

```
arr_allTestRatings=allTestRatings.collect()#returns an array
```

```
type(arr_allTestRatings)
```

```
list
```

MovieLens 100K – Performance Evaluation

```
# FIND MSE with for loop
sum = 0  #variable to store the summation of differences
n = len(arr_allPredictedRatings) #finding total number of items in list
for i in range (0,n):  #looping through each element of the list
    difference =arr_allTestRatings[i]-arr_allPredictedRatings[i]  #finding the d
ifference between observed and predicted value
    squared_difference = difference**2  #taking square of the differene
    sum = sum + squared_difference  #taking a sum of all the differences
MSE = sum/n  #dividing summation by total values to obtain average
print ("The Mean Square Error is: " , MSE)
```

```
The Mean Square Error is:  2.348931916418124
```

- You can calculate MSE with single line by using functions in ***numpy*** library.
- Also it is possible to use "join" operation to crate structure that gathers prediction and original values.

```
[((276, 796), (1.0, 2.496714933453478)),
 ((127, 229), (5.0, 2.772037864942624)),
 ((276, 54), (3.0, 2.742805088560613)),
 ((95, 625), (4.0, 4.2333791481715)),
 ((119, 1153), (5.0, 3.885016019369801 6)),
 ((286, 208), (4.0, 3.440506956009861 6)),
 ((299, 111), (3.0, 2.687430151689176)),
 ((85, 427), (3.0, 4.376090277425813 5))
```