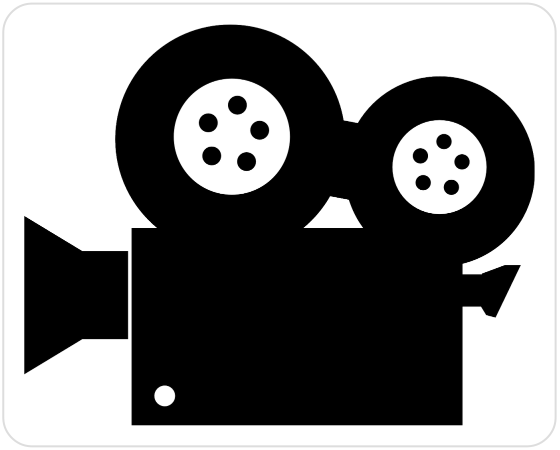
databricks cs110_lab2_als_prediction

Predicting Movie Ratings



One of the most common uses of big data is to predict what users want. This allows Google to show you relevant ads, Amazon to recommend relevant products, and Netflix to recommend movies that you might like. This lab will demonstrate how we can use Apache Spark to recommend movies to a user. We will start with some basic techniques, and then use the Spark ML (https://spark.apache.org/docs/1.6.2/api/python/pyspark.ml.html) library's Alternating Least Squares method to make more sophisticated predictions.

For this lab, we will use a subset dataset of 20 million ratings. This dataset is pre-mounted on Databricks and is from the MovieLens stable benchmark rating dataset (http://grouplens.org/datasets/movielens/). However, the same code you write will also work on the full dataset (though running with the full dataset on Community Edition is likely to take quite a long time).

In this lab:

- *Part 0*: Preliminaries
- *Part 1*: Basic Recommendations
- *Part 2*: Collaborative Filtering
- *Part 3*: Predictions for Yourself

As mentioned during the first Learning Spark lab, think carefully before calling `collect()` on any datasets. When you are using a small dataset, calling `collect()` and then using Python to get a sense for the data locally (in the driver program) will work fine, but this will not work when you are using a large dataset that doesn't fit in memory on one machine. Solutions that call `collect()` and do local analysis that could have been done with Spark will likely fail in the autograder and not receive full credit.

```
labVersion = 'cs110x.lab2-1.0.0'
```

Code

This assignment can be completed using basic Python and pySpark DataFrame Transformations and Actions. Libraries other than math are not necessary. With the exception of the ML functions that we introduce in this assignment, you should be able to complete all parts of this homework using only the Spark functions you have used in prior lab exercises (although you are welcome to use more features of Spark if you like!).

We'll be using motion picture data, the same data last year's CS100.1x used. However, in this course, we're using DataFrames, rather than RDDs.

The following cell defines the locations of the data files. If you want to run an exported version of this lab on your own machine (i.e., outside of Databricks), you'll need to download your own copy of the 20-million movie data set, and you'll need to adjust the paths, below.

To Do: Run the following cell.

```
import os
from databricks_test_helper import Test

dbfs_dir = '/databricks-datasets/cs110x/ml-20m/data-001'
ratings_filename = dbfs_dir + '/ratings.csv'
movies_filename = dbfs_dir + '/movies.csv'

# The following line is here to enable this notebook to be exported as source
and
# run on a local machine with a local copy of the files. Just change the
dbfs_dir,
# above.
if os.path.sep != '/':
  # Handle Windows.
  ratings_filename = ratings_filename.replace('/', os.path.sep)
  movie_filename = movie_filename.replace('/', os.path.sep)
```

Part 0: Preliminaries

We read in each of the files and create a DataFrame consisting of parsed lines.

The 20-million movie sample

The 20-million movie sample consists of CSV files (with headers), so there's no need to parse the files manually, as Spark CSV can do the job.

First, let's take a look at the directory containing our files.

```
display(dbutils.fs.ls(dbfs_dir))
```

path
dbfs:/databricks-datasets/cs110x/ml-20m/data-001/README.txt
dbfs:/databricks-datasets/cs110x/ml-20m/data-001/links.csv
dbfs:/databricks-datasets/cs110x/ml-20m/data-001/links.csv.gz
dbfs:/databricks-datasets/cs110x/ml-20m/data-001/movies.csv
dbfs:/databricks-datasets/cs110x/ml-20m/data-001/movies.csv.gz
dbfs:/databricks-datasets/cs110x/ml-20m/data-001/ratings.csv
dbfs:/databricks-datasets/cs110x/ml-20m/data-001/ratings.csv.gz

CPU vs I/O tradeoff

Note that we have both compressed files (ending in `.gz`) and uncompressed files. We have a CPU vs. I/O tradeoff here. If I/O is the bottleneck, then we want to process the compressed files and pay the extra CPU overhead. If CPU is the bottleneck, then it makes more sense to process the uncompressed files.

We've done some experiments, and we've determined that CPU is more of a bottleneck than I/O, on Community Edition. So, we're going to process the uncompressed data. In addition, we're going to speed things up further by specifying the DataFrame schema explicitly. (When the Spark CSV adapter infers the schema from a CSV file, it has to make an extra pass over the file. That'll slow things down here, and it isn't really necessary.)

To Do: Run the following cell, which will define the schemas.

```
from pyspark.sql.types import *

ratings_df_schema = StructType(
  [StructField('userId', IntegerType()),
   StructField('movieId', IntegerType()),
   StructField('rating', DoubleType())]
)
movies_df_schema = StructType(
  [StructField('ID', IntegerType()),
   StructField('title', StringType())]
)
```

Load and Cache

The Databricks File System (DBFS) sits on top of S3. We're going to be accessing this data a lot. Rather than read it over and over again from S3, we'll cache both the movies DataFrame and the ratings DataFrame in memory.

To Do: Run the following cell to load and cache the data. Please be patient: The code takes about 30 seconds to run.

```
from pyspark.sql.functions import regexp_extract
from pyspark.sql.types import *

raw_ratings_df =
sqlContext.read.format('com.databricks.spark.csv').options(header=True,
inferSchema=False).schema(ratings_df_schema).load(ratings_filename)
ratings_df = raw_ratings_df.drop('Timestamp')

raw_movies_df =
sqlContext.read.format('com.databricks.spark.csv').options(header=True,
inferSchema=False).schema(movies_df_schema).load(movies_filename)
movies_df = raw_movies_df.drop('Genres').withColumnRenamed('movieId', 'ID')

ratings_df.cache()
movies_df.cache()

assert ratings_df.is_cached
assert movies_df.is_cached

raw_ratings_count = raw_ratings_df.count()
ratings_count = ratings_df.count()
raw_movies_count = raw_movies_df.count()
movies_count = movies_df.count()

print 'There are %s ratings and %s movies in the datasets' % (ratings_count,
movies_count)
print 'Ratings:'
ratings_df.show(3)
print 'Movies:'
movies_df.show(3, truncate=False)

assert raw_ratings_count == ratings_count
assert raw_movies_count == movies_count
```

```
There are 20000263 ratings and 27278 movies in the datasets
Ratings:
+------+-------+------+
|userId|movieId|rating|
+------+-------+------+
|     1|      2|   3.5|
|     1|     29|   3.5|
|     1|     32|   3.5|
+------+-------+------+
only showing top 3 rows

Movies:
+---+-----------------------+
|ID |title                  |
+---+-----------------------+
|1  |Toy Story (1995)       |
|2  |Jumanji (1995)         |
|3  |Grumpier Old Men (1995)|
+---+-----------------------+
only showing top 3 rows
```

Next, let's do a quick verification of the data.

To do: Run the following cell. It should run without errors.

```
assert ratings_count == 20000263
assert movies_count == 27278
assert movies_df.filter(movies_df.title == 'Toy Story (1995)').count() == 1
assert ratings_df.filter((ratings_df.userId == 6) & (ratings_df.movieId == 1) &
(ratings_df.rating == 5.0)).count() == 1
```

Let's take a quick look at some of the data in the two DataFrames.

To Do: Run the following two cells.

```
display(movies_df)
```

ID	title
1	Toy Story (1995)
2	Jumanji (1995)
3	Grumpier Old Men (1995)
4	Waiting to Exhale (1995)
5	Father of the Bride Part II (1995)
6	Heat (1995)
7	Sabrina (1995)
8	Tom and Huck (1995)
9	Sudden Death (1995)

Showing the first 1000 rows.

```
display(ratings_df)
```

userId	moviel
1	2
1	29
1	32
1	47
1	50
1	112
1	151
1	223
1	253

Showing the first 1000 rows.

Part 1: Basic Recommendations

One way to recommend movies is to always recommend the movies with the highest average rating. In this part, we will use Spark to find the name, number of ratings, and the average rating of the 20 movies with the highest average rating and at least 500 reviews. We want to filter our movies with high ratings but greater than or equal to 500 reviews because movies with few reviews may not have broad appeal to everyone.

(1a) Movies with Highest Average Ratings

Let's determine the movies with the highest average ratings.

The steps you should perform are:

1. Recall that the `ratings_df` contains three columns:
   - The ID of the user who rated the film
   - the ID of the movie being rated

   - and the rating.

   First, transform `ratings_df` into a second DataFrame, `movie_ids_with_avg_ratings`, with the following columns:
   - The movie ID
   - The number of ratings for the movie
   - The average of all the movie's ratings

2. Transform `movie_ids_with_avg_ratings` to another DataFrame, `movie_names_with_avg_ratings_df` that adds the movie name to each row. `movie_names_with_avg_ratings_df` will contain these columns:
   - The movie ID
   - The movie name
   - The number of ratings for the movie
   - The average of all the movie's ratings

   Hint: You'll need to do a join.

You should end up with something like the following:

```
 movie_ids_with_avg_ratings_df:
+-------+-----+------------------+
|movieId|count|average           |
+-------+-----+------------------+
|1831   |7463 |2.5785207021305103|
|431    |8946 |3.695059244355019 |
|631    |2193 |2.7273141814865483|
+-------+-----+------------------+
only showing top 3 rows

movie_names_with_avg_ratings_df:
+-------+-----------------------------+-----+-------+
|average|title                        |count|movieId|
+-------+-----------------------------+-----+-------+
|5.0    |Ella Lola, a la Trilby (1898)|1    |94431  |
|5.0    |Serving Life (2011)          |1    |129034 |
|5.0    |Diplomatic Immunity (2009? ) |1    |107434 |
+-------+-----------------------------+-----+-------+
only showing top 3 rows
```

```
# TODO: Replace <FILL_IN> with appropriate code
from pyspark.sql import functions as F

# From ratingsDF, create a movie_ids_with_avg_ratings_df that combines the two
DataFrames
movie_ids_with_avg_ratings_df =
ratings_df.groupBy('movieId').agg(F.count(ratings_df.rating).alias("count"),
F.avg(ratings_df.rating).alias("average"))
print 'movie_ids_with_avg_ratings_df:'
movie_ids_with_avg_ratings_df.show(3, truncate=False)

# Note: movie_names_df is a temporary variable, used only to separate the steps
necessary
# to create the movie_names_with_avg_ratings_df DataFrame.
movie_names_df =
movie_ids_with_avg_ratings_df.join(movies_df,movie_ids_with_avg_ratings_df["mov
ieId"]==movies_df["Id"])
movie_names_with_avg_ratings_df = movie_names_df.drop("Id")

print 'movie_names_with_avg_ratings_df:'
movie_names_with_avg_ratings_df.show(3, truncate=False)
```

```
movie_ids_with_avg_ratings_df:
+-------+-----+------------------+
|movieId|count|average           |
+-------+-----+------------------+
|1831   |7463 |2.5785207021305103|
|431    |8946 |3.695059244355019 |
|631    |2193 |2.7273141814865483|
+-------+-----+------------------+
only showing top 3 rows

movie_names_with_avg_ratings_df:
+-------+-----+------------------+-----------------------------+
|movieId|count|average           |title                        |
+-------+-----+------------------+-----------------------------+
|1831   |7463 |2.5785207021305103|Lost in Space (1998)         |
|431    |8946 |3.695059244355019 |Carlito's Way (1993)         |
|631    |2193 |2.7273141814865483|All Dogs Go to Heaven 2 (1996)|
+-------+-----+------------------+-----------------------------+
only showing top 3 rows
```

```
# TEST Movies with Highest Average Ratings (1a)
Test.assertEquals(movie_ids_with_avg_ratings_df.count(), 26744,
                'incorrect movie_ids_with_avg_ratings_df.count() (expected
26744)')
movie_ids_with_ratings_take_ordered =
movie_ids_with_avg_ratings_df.orderBy('MovieID').take(3)
_take_0 = movie_ids_with_ratings_take_ordered[0]
_take_1 = movie_ids_with_ratings_take_ordered[1]
_take_2 = movie_ids_with_ratings_take_ordered[2]
Test.assertTrue(_take_0[0] == 1 and _take_0[1] == 49695,
                'incorrect count of ratings for movie with ID {0} (expected
49695)'.format(_take_0[0]))
Test.assertEquals(round(_take_0[2], 2), 3.92, "Incorrect average for movie ID
{0}. Expected 3.92".format(_take_0[0]))

Test.assertTrue(_take_1[0] == 2 and _take_1[1] == 22243,
                'incorrect count of ratings for movie with ID {0} (expected
22243)'.format(_take_1[0]))
Test.assertEquals(round(_take_1[2], 2), 3.21, "Incorrect average for movie ID
{0}. Expected 3.21".format(_take_1[0]))

Test.assertTrue(_take_2[0] == 3 and _take_2[1] == 12735,
                'incorrect count of ratings for movie with ID {0} (expected
12735)'.format(_take_2[0]))
Test.assertEquals(round(_take_2[2], 2), 3.15, "Incorrect average for movie ID
{0}. Expected 3.15".format(_take_2[0]))


Test.assertEquals(movie_names_with_avg_ratings_df.count(), 26744,
                  'incorrect movie_names_with_avg_ratings_df.count() (expected
26744)')
movie_names_with_ratings_take_ordered =
movie_names_with_avg_ratings_df.orderBy(['average', 'title']).take(3)
result = [(r['average'], r['title'], r['count'], r['movieId']) for r in
movie_names_with_ratings_take_ordered]
Test.assertEquals(result,
                  [(0.5, u'13 Fighting Men (1960)', 1, 109355),
                   (0.5, u'20 Years After (2008)', 1, 131062),
                   (0.5, u'3 Holiday Tails (Golden Christmas 2: The Second
Tail, A) (2011)', 1, 111040)],
                  'incorrect top 3 entries in movie_names_with_avg_ratings_df')
```

```
1 test passed.
1 test passed.
1 test passed.
1 test passed.
1 test passed.
```

```
1 test passed.
1 test passed.
1 test passed.
1 test passed.
```

(1b) Movies with Highest Average Ratings and at least 500 reviews

Now that we have a DataFrame of the movies with highest average ratings, we can use Spark to determine the 20 movies with highest average ratings and at least 500 reviews.

Add a single DataFrame transformation (in place of `<FILL_IN>` , below) to limit the results to movies with ratings from at least 500 people.

```
# TODO: Replace <FILL IN> with appropriate code
# TODO: Replace <FILL IN> with appropriate code
movies_with_500_ratings_or_more =
movie_names_with_avg_ratings_df.where(movie_names_with_avg_ratings_df["count"]>
=500)
print 'Movies with highest ratings:'
movies_with_500_ratings_or_more.show(20, truncate=False)
```

```
Movies with highest ratings:
+-------+-----+------------------+-----------------------------------------+
|movieId|count|average           |title                                    |
+-------+-----+------------------+-----------------------------------------+
|1831   |7463 |2.5785207021305103|Lost in Space (1998)                     |
|431    |8946 |3.695059244355019 |Carlito's Way (1993)                     |
|631    |2193 |2.7273141814865483|All Dogs Go to Heaven 2 (1996)           |
|231    |32085|2.9524700015583605|Dumb & Dumber (Dumb and Dumber) (1994)   |
|32031  |1777 |3.270962296004502 |Robots (2005)                            |
|45431  |2059 |3.416221466731423 |Over the Hedge (2006)                    |
|2231   |6759 |3.727400503032993 |Rounders (1998)                          |
|2431   |6045 |3.109677419354839 |Patch Adams (1998)                       |
|4031   |1047 |2.89207258834766  |All the Pretty Horses (2000)             |
|31     |9435 |3.250344462109168 |Dangerous Minds (1995)                   |
|1231   |7902 |3.9474816502151353|Right Stuff, The (1983)                  |
|3831   |1564 |3.6620843989769822|Saving Grace (2000)                      |
|27831  |2883 |3.7835587929240373|Layer Cake (2004)                        |
|1031   |4227 |3.329548142890939 |Bedknobs and Broomsticks (1971)          |
|4231   |996  |3.0737951807228914|Someone Like You (2001)                  |
|1431   |2931 |2.571989082224497 |Beverly Hills Ninja (1997)               |
|39231  |1097 |3.1180492251595258|Elizabethtown (2005)                     |
|80831  |581  |3.621342512908778 |Let Me In (2010)                         |
|32     |44980|3.8980546909737663|Twelve Monkeys (a.k.a. 12 Monkeys) (1995)|
|832    |14347|3.4579354568899423|Ransom (1996)                            |
+-------+-----+------------------+-----------------------------------------+
only showing top 20 rows
```

```
# TEST Movies with Highest Average Ratings and at least 500 Reviews (1b)

Test.assertEquals(movies_with_500_ratings_or_more.count(), 4489,
                  'incorrect movies_with_500_ratings_or_more.count(). Expected
4489.')
top_20_results = [(r['average'], r['title'], r['count']) for r in
movies_with_500_ratings_or_more.orderBy(F.desc('average')).take(20)]

Test.assertEquals(top_20_results,
                  [(4.446990499637029, u'Shawshank Redemption, The (1994)',
63366),
                   (4.364732196832306, u'Godfather, The (1972)', 41355),
                   (4.334372207803259, u'Usual Suspects, The (1995)', 47006),
                   (4.310175010988133, u"Schindler's List (1993)", 50054),
                   (4.275640557704942, u'Godfather: Part II, The (1974)',
27398),
                   (4.2741796572216, u'Seven Samurai (Shichinin no samurai)
(1954)', 11611),
                   (4.271333600779414, u'Rear Window (1954)', 17449),
                   (4.263182346109176, u'Band of Brothers (2001)', 4305),
                   (4.258326830670664, u'Casablanca (1942)', 24349),
                   (4.256934865900383, u'Sunset Blvd. (a.k.a. Sunset Boulevard)
(1950)', 6525),
                   (4.24807897901911, u"One Flew Over the Cuckoo's Nest
(1975)", 29932),
                   (4.247286821705426, u'Dr. Strangelove or: How I Learned to
Stop Worrying and Love the Bomb (1964)', 23220),
                   (4.246001523229246, u'Third Man, The (1949)', 6565),
                   (4.235410064157069, u'City of God (Cidade de Deus) (2002)',
12937),
                   (4.2347902097902095, u'Lives of Others, The (Das leben der
Anderen) (2006)', 5720),
                   (4.233538107122288, u'North by Northwest (1959)', 15627),
                   (4.2326233183856505, u'Paths of Glory (1957)', 3568),
                   (4.227123123722136, u'Fight Club (1999)', 40106),
                   (4.224281931146873, u'Double Indemnity (1944)', 4909),
                   (4.224137931034483, u'12 Angry Men (1957)', 12934)],
                  'Incorrect top 20 movies with 500 or more ratings')
```

```
1 test passed.
1 test passed.
```

Using a threshold on the number of reviews is one way to improve the recommendations, but there are many other good ways to improve quality. For example, you could weight ratings by the number of ratings.

Part 2: Collaborative Filtering

In this course, you have learned about many of the basic transformations and actions that Spark allows us to apply to distributed datasets. Spark also exposes some higher level functionality; in particular, Machine Learning using a component of Spark called MLlib (http://spark.apache.org/docs/1.6.2/mllib-guide.html). In this part, you will learn how to use MLlib to make personalized movie recommendations using the movie data we have been analyzing.

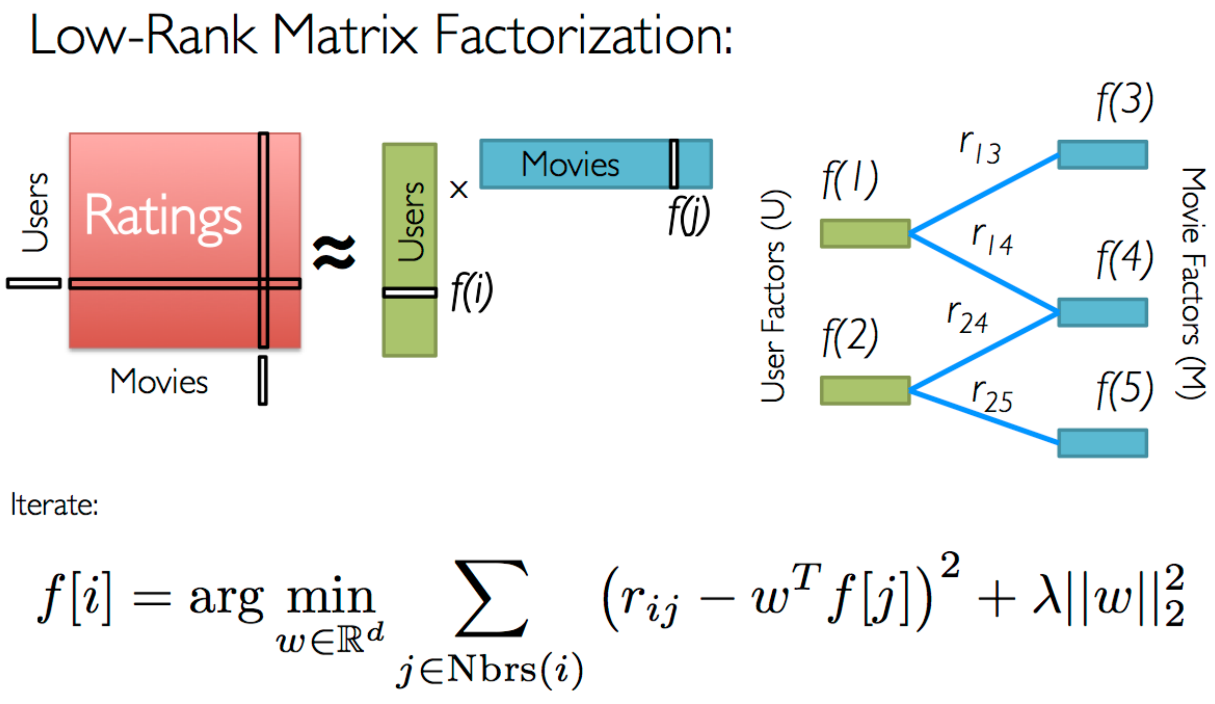
We are going to use a technique called collaborative filtering

(https://en.wikipedia.org/?title=Collaborative_filtering). Collaborative filtering is a method of making automatic predictions (filtering) about the interests of a user by collecting preferences or taste information from many users (collaborating). The underlying assumption of the collaborative filtering approach is that if a person A has the same opinion as a person B on an issue, A is more likely to have B's opinion on a different issue x than to have the opinion on x of a person chosen randomly. You can read more about collaborative filtering here (http://recommender-systems.org/collaborative-filtering/).

The image at the right (from Wikipedia (https://en.wikipedia.org/?title=Collaborative_filtering)) shows an example of predicting of the user's rating using collaborative filtering. At first, people rate different items (like videos, images, games). After that, the system is making predictions about a user's rating for an item, which the user has not rated yet. These predictions are built upon the existing ratings of other users, who have similar ratings with the active user. For instance, in the image below the system has made a prediction, that the active user will not like the video.

For movie recommendations, we start with a matrix whose entries are movie ratings by users (shown in red in the diagram below). Each column represents a user (shown in green) and each row represents a particular movie (shown in blue).

Since not all users have rated all movies, we do not know all of the entries in this matrix, which is precisely why we need collaborative filtering. For each user, we have ratings for only a subset of the movies. With collaborative filtering, the idea is to approximate the ratings matrix by factorizing it as the product of two matrices: one that describes properties of each user (shown in green), and one that describes properties of each movie (shown in blue).



$$f[i] = \arg\min_{w \in \mathbb{R}^d} \sum_{j \in \text{Nbrs}(i)} \left(r_{ij} - w^T f[j]\right)^2 + \lambda ||w||_2^2$$

We want to select these two matrices such that the error for the users/movie pairs where we know the correct ratings is minimized. The Alternating Least Squares (https://en.wikiversity.org/wiki/Least-Squares_Method) algorithm does this by first randomly filling the users matrix with values and then optimizing the value of the movies such that the error is minimized. Then, it holds the movies matrix constant and optimizes the value of the user's matrix. This alternation between which matrix to optimize is the reason for the "alternating" in the name.

This optimization is what's being shown on the right in the image above. Given a fixed set of user factors (i.e., values in the users matrix), we use the known ratings to find the best values for the movie factors using the optimization written at the bottom of the figure. Then we "alternate" and pick the best user factors given fixed movie factors.

For a simple example of what the users and movies matrices might look like, check out the videos from Lecture 2 (https://courses.edx.org/courses/course-v1:BerkeleyX+CS110x+2T2016/courseware/9d251397874d4f0b947b606c81ccf83c/3cf or the slides from Lecture 8 (https://d37djvu3ytnwxt.cloudfront.net/assets/courseware/v1/fb269ff9a53b669a46d59e v1:BerkeleyX+CS110x+2T2016+type@asset+block/Lecture2s.pdf)

(2a) Creating a Training Set

Before we jump into using machine learning, we need to break up the `ratings_df` dataset into three pieces:

- A training set (DataFrame), which we will use to train models
- A validation set (DataFrame), which we will use to choose the best model
- A test set (DataFrame), which we will use for our experiments

To randomly split the dataset into the multiple groups, we can use the pySpark randomSplit() (http://spark.apache.org/docs/1.6.2/api/python/pyspark.sql.html#pyspark.sql.DataFram transformation. `randomSplit()` takes a set of splits and a seed and returns multiple DataFrames.

```
# TODO: Replace <FILL_IN> with the appropriate code.

# We'll hold out 60% for training, 20% of our data for validation, and leave
20% for testing
seed = 1800009193L
(split_60_df, split_a_20_df, split_b_20_df) =
ratings_df.randomSplit([0.6,0.2,0.2],seed)

# Let's cache these datasets for performance
training_df = split_60_df.cache()
validation_df = split_a_20_df.cache()
test_df = split_b_20_df.cache()

print('Training: {0}, validation: {1}, test: {2}\n'.format(
  training_df.count(), validation_df.count(), test_df.count())
)
training_df.show(3)
validation_df.show(3)
test_df.show(3)
```

```
Training: 12001389, validation: 4003694, test: 3995180

+------+-------+------+
|userId|movieId|rating|
+------+-------+------+
|     1|      2|   3.5|
|     1|     29|   3.5|
|     1|     47|   3.5|
+------+-------+------+
only showing top 3 rows

+------+-------+------+
|userId|movieId|rating|
+------+-------+------+
|     1|     32|   3.5|
|     1|    253|   4.0|
|     1|    293|   4.0|
+------+-------+------+
only showing top 3 rows

+------+-------+------+
|userId|movieId|rating|
+------+-------+------+
|     1|    112|   3.5|
|     1|    151|   4.0|
|     1|    318|   4.0|
+------+-------+------+
only showing top 3 rows
```

```
# TEST Creating a Training Set (2a)
Test.assertEquals(training_df.count(), 12001389, "Incorrect training_df count.
Expected 12001389")
Test.assertEquals(validation_df.count(), 4003694, "Incorrect validation_df
count. Expected 4003694")
Test.assertEquals(test_df.count(), 3995180, "Incorrect test_df count. Expected
3995180")

Test.assertEquals(training_df.filter((ratings_df.userId == 1) &
(ratings_df.movieId == 5952) & (ratings_df.rating == 5.0)).count(), 1)
Test.assertEquals(training_df.filter((ratings_df.userId == 1) &
(ratings_df.movieId == 1193) & (ratings_df.rating == 3.5)).count(), 1)
Test.assertEquals(training_df.filter((ratings_df.userId == 1) &
(ratings_df.movieId == 1196) & (ratings_df.rating == 4.5)).count(), 1)

Test.assertEquals(validation_df.filter((ratings_df.userId == 1) &
(ratings_df.movieId == 296) & (ratings_df.rating == 4.0)).count(), 1)
Test.assertEquals(validation_df.filter((ratings_df.userId == 1) &
(ratings_df.movieId == 32) & (ratings_df.rating == 3.5)).count(), 1)
Test.assertEquals(validation_df.filter((ratings_df.userId == 1) &
(ratings_df.movieId == 6888) & (ratings_df.rating == 3.0)).count(), 1)

Test.assertEquals(test_df.filter((ratings_df.userId == 1) & (ratings_df.movieId
== 4993) & (ratings_df.rating == 5.0)).count(), 1)
Test.assertEquals(test_df.filter((ratings_df.userId == 1) & (ratings_df.movieId
== 4128) & (ratings_df.rating == 4.0)).count(), 1)
Test.assertEquals(test_df.filter((ratings_df.userId == 1) & (ratings_df.movieId
== 4915) & (ratings_df.rating == 3.0)).count(), 1)
```

```
1 test passed.
1 test passed.
1 test passed.
1 test passed.
1 test passed.
1 test passed.
1 test passed.
1 test passed.
1 test passed.
1 test passed.
1 test passed.
1 test passed.
```

After splitting the dataset, your training set has about 12 million entries and the validation and test sets each have about 4 million entries. (The exact number of entries in each dataset varies slightly due to the random nature of the `randomSplit()` transformation.)

(2b) Alternating Least Squares

In this part, we will use the Apache Spark ML Pipeline implementation of Alternating Least Squares, ALS (http://spark.apache.org/docs/1.6.2/api/python/pyspark.ml.html#pyspark.ml.recommenc ALS takes a training dataset (DataFrame) and several parameters that control the model creation process. To determine the best values for the parameters, we will use ALS to train several models, and then we will select the best model and use the parameters from that model in the rest of this lab exercise.

The process we will use for determining the best model is as follows:

1. Pick a set of model parameters. The most important parameter to model is the *rank*, which is the number of columns in the Users matrix (green in the diagram above) or the number of rows in the Movies matrix (blue in the diagram above). In general, a lower rank will mean higher error on the training dataset, but a high rank may lead to overfitting (https://en.wikipedia.org/wiki/Overfitting). We will train models with ranks of 4, 8, and 12 using the `training_df` dataset.

2. Set the appropriate parameters on the `ALS` object:

- The "User" column will be set to the values in our `userId` DataFrame column.
- The "Item" column will be set to the values in our `movieId` DataFrame column.
- The "Rating" column will be set to the values in our `rating` DataFrame column.
- We'll using a regularization parameter of 0.1.

Note: Read the documentation for the ALS (http://spark.apache.org/docs/1.6.2/api/python/pyspark.ml.html#pyspark.ml.recomm class **carefully**. It will help you accomplish this step.

3. Have the ALS output transformation (i.e., the result of ALS.fit() (http://spark.apache.org/docs/1.6.2/api/python/pyspark.ml.html#pyspark.ml.recomm produce a *new* column called "prediction" that contains the predicted value.

4. Create multiple models using ALS.fit() (http://spark.apache.org/docs/1.6.2/api/python/pyspark.ml.html#pyspark.ml.recomm one for each of our rank values. We'll fit against the training data set (`training_df`).

5. For each model, we'll run a prediction against our validation data set (`validation_df`) and check the error.

6. We'll keep the model with the best error rate.

Why are we doing our own cross-validation?

A challenge for collaborative filtering is how to provide ratings to a new user (a user who has not provided *any* ratings at all). Some recommendation systems choose to provide new users with a set of default ratings (e.g., an average value across all ratings), while others choose to provide no ratings for new users. Spark's ALS algorithm yields a NaN (`Not a Number`) value when asked to provide a rating for a new user.

Using the ML Pipeline's CrossValidator (http://spark.apache.org/docs/1.6.2/api/python/pyspark.ml.html#pyspark.ml.tuning.Cros with ALS is thus problematic, because cross validation involves dividing the training data into a set of folds (e.g., three sets) and then using those folds for testing and evaluating the parameters during the parameter grid search process. It is likely that some of the folds will contain users that are not in the other folds, and, as a result, ALS produces NaN values for those new users. When the CrossValidator uses the Evaluator (RMSE) to compute an error metric, the RMSE algorithm will return NaN. This will make *all* of the parameters in the parameter grid appear to be equally good (or bad).

You can read the discussion on Spark JIRA 14489 (https://issues.apache.org/jira/browse/SPARK-14489) about this issue. There are proposed workarounds of having ALS provide default values or having RMSE drop NaN values. Both introduce potential issues. We have chosen to have RMSE drop NaN values. While this does not solve the underlying issue of ALS not predicting a value for a new user, it does provide some evaluation value. We manually implement the parameter grid search process using a for loop (below) and remove the NaN values before using RMSE.

For a production application, you would want to consider the tradeoffs in how to handle new users.

Note: This cell will likely take a couple of minutes to run.

```
# TODO: Replace <FILL IN> with appropriate code
# This step is broken in ML Pipelines:
https://issues.apache.org/jira/browse/SPARK-14489
from pyspark.ml.recommendation import ALS

# Let's initialize our ALS learner
als = ALS()

# Now we set the parameters for the method
als.setMaxIter(5)\
   .setSeed(seed)\
   .setRegParam(0.1)\
   .setUserCol("userId").setItemCol("movieId").setRatingCol("rating")

# Now let's compute an evaluation metric for our test dataset
from pyspark.ml.evaluation import RegressionEvaluator

# Create an RMSE evaluator using the label and predicted columns
reg_eval = RegressionEvaluator(predictionCol="prediction", labelCol="rating",
metricName="rmse")

tolerance = 0.03
ranks = [4, 8, 12]
errors = [0, 0, 0]
models = [0, 0, 0]
err = 0
min_error = float('inf')
best_rank = -1
for rank in ranks:
  # Set the rank here:
  als.setRank(rank)
  # Create the model with these parameters.
  model = als.fit(training_df)
  # Run the model to create a prediction. Predict against the validation_df.
  predict_df = model.transform(validation_df)

  # Remove NaN values from prediction (due to SPARK-14489)
  predicted_ratings_df = predict_df.filter(predict_df.prediction !=
float('nan'))

  # Run the previously created RMSE evaluator, reg_eval, on the
predicted_ratings_df DataFrame
  error = reg_eval.evaluate(predicted_ratings_df)
  errors[err] = error
  models[err] = model
  print 'For rank %s the RMSE is %s' % (rank, error)
  if error < min_error:
      min_error = error
      best_rank = err
    err += 1

als.setRank(ranks[best_rank])
print 'The best model was trained with rank %s' % ranks[best_rank]
my_model = models[best_rank]
```

```
For rank 4 the RMSE is 0.82825406832
For rank 8 the RMSE is 0.816154128069
For rank 12 the RMSE is 0.810037726846
The best model was trained with rank 12
```

```
# TEST
Test.assertEquals(round(min_error, 2), 0.81, "Unexpected value for best RMSE.
Expected rounded value to be 0.81. Got {0}".format(round(min_error, 2)))
Test.assertEquals(ranks[best_rank], 12, "Unexpected value for best rank.
Expected 12. Got {0}".format(ranks[best_rank]))
Test.assertEqualsHashed(als.getItemCol(),
"18f0e2357f8829fe809b2d95bc1753000dd925a6", "Incorrect choice of {0} for ALS
item column.".format(als.getItemCol()))
Test.assertEqualsHashed(als.getUserCol(),
"db36668fa9a19fde5c9676518f9e86c17cabf65a", "Incorrect choice of {0} for ALS
user column.".format(als.getUserCol()))
Test.assertEqualsHashed(als.getRatingCol(),
"3c2d687ef032e625aa4a2b1cfca9751d2080322c", "Incorrect choice of {0} for ALS
rating column.".format(als.getRatingCol()))
```

```
1 test passed.
1 test passed.
1 test passed.
1 test passed.
1 test passed.
```

(2c) Testing Your Model

So far, we used the `training_df` and `validation_df` datasets to select the best model. Since we used these two datasets to determine what model is best, we cannot use them to test how good the model is; otherwise, we would be very vulnerable to overfitting (https://en.wikipedia.org/wiki/Overfitting). To decide how good our model is, we need to use the `test_df` dataset. We will use the `best_rank` you determined in part (2b) to create a model for predicting the ratings for the test dataset and then we will compute the RMSE.

The steps you should perform are:

- Run a prediction, using `my_model` as created above, on the test dataset (`test_df`), producing a new `predict_df` DataFrame.
- Filter out unwanted NaN values (necessary because of a bug in Spark (https://issues.apache.org/jira/browse/SPARK-14489)). We've supplied this piece of code for you.
- Use the previously created RMSE evaluator, `reg_eval` to evaluate the filtered DataFrame.

```
# TODO: Replace <FILL_IN> with the appropriate code
# In ML Pipelines, this next step has a bug that produces unwanted NaN values.
We
# have to filter them out. See https://issues.apache.org/jira/browse/SPARK-
14489
predict_df = my_model.transform(test_df)

# Remove NaN values from prediction (due to SPARK-14489)
predicted_test_df = predict_df.filter(predict_df.prediction != float('nan'))

# Run the previously created RMSE evaluator, reg_eval, on the predicted_test_df
DataFrame
test_RMSE = reg_eval.evaluate(predicted_test_df)

print('The model had a RMSE on the test set of {0}'.format(test_RMSE))
```

```
The model had a RMSE on the test set of 0.809624038485
```

```
# TEST Testing Your Model (2c)
Test.assertTrue(abs(test_RMSE - 0.809624038485) < tolerance, 'incorrect
test_RMSE: {0:.11f}'.format(test_RMSE))
```

```
1 test passed.
```

(2d) Comparing Your Model

Looking at the RMSE for the results predicted by the model versus the values in the test set is one way to evalute the quality of our model. Another way to evaluate the model is to evaluate the error from a test set where every rating is the average rating for the training set.

The steps you should perform are:

- Use the `training_df` to compute the average rating across all movies in that training dataset.
- Use the average rating that you just determined and the `test_df` to create a DataFrame (`test_for_avg_df`) with a `prediction` column containing the average rating. **HINT**: You'll want to use the `lit()` function, from `pyspark.sql.functions` , available here as `F.lit()` .
- Use our previously created `reg_eval` object to evaluate the `test_for_avg_df` and calculate the RMSE.

```
# TODO: Replace <FILL_IN> with the appropriate code.
# Compute the average rating
avg_rating_df = training_df.agg(F.avg("rating"))

# Extract the average rating value. (This is row 0, column 0.)
training_avg_rating = avg_rating_df.collect()[0][0]

print('The average rating for movies in the training set is
{0}'.format(training_avg_rating))

# Add a column with the average rating
test_for_avg_df = training_df.withColumn('prediction',
F.lit(training_avg_rating))

# Run the previously created RMSE evaluator, reg_eval, on the test_for_avg_df
DataFrame
test_avg_RMSE = reg_eval.evaluate(test_for_avg_df)

print("The RMSE on the average set is {0}".format(test_avg_RMSE))
```

```
The average rating for movies in the training set is 3.52547984237
The RMSE on the average set is 1.0519743756
```

```
# TEST Comparing Your Model (2d)
Test.assertTrue(abs(training_avg_rating - 3.52547984237) < 0.000001,
                'incorrect training_avg_rating (expected 3.52547984237):
{0:.11f}'.format(training_avg_rating))
Test.assertTrue(abs(test_avg_RMSE - 1.05190953037) < 0.000001,
                'incorrect test_avg_RMSE (expected 1.0519743756):
{0:.11f}'.format(test_avg_RMSE))
```

```
1 test passed.
1 test failed. incorrect test_avg_RMSE (expected 1.0519743756): 1.05197437560
```

You now have code to predict how users will rate movies!

Part 3: Predictions for Yourself

The ultimate goal of this lab exercise is to predict what movies to recommend to yourself. In order to do that, you will first need to add ratings for yourself to the `ratings_df` dataset.

(3a) Your Movie Ratings

To help you provide ratings for yourself, we have included the following code to list the names and movie IDs of the 50 highest-rated movies from `movies_with_500_ratings_or_more` which we created in part 1 the lab.

```
print 'Most rated movies:'
print '(average rating, movie name, number of reviews, movie ID)'
display(movies_with_500_ratings_or_more.orderBy(movies_with_500_ratings_or_more
['average'].desc()).take(50))
```

movieId	count	average
318	63366	4.446990499637029
858	41355	4.364732196832306
50	47006	4.334372207803259
527	50054	4.310175010988133
1221	27398	4.275640557704942
2019	11611	4.2741796572216
904	17449	4.271333600779414
7502	4305	4.263182346109176
912	24349	4.258326830670664

The user ID 0 is unassigned, so we will use it for your ratings. We set the variable `my_user_ID` to 0 for you. Next, create a new DataFrame called `my_ratings_df`, with your ratings for at least 10 movie ratings. Each entry should be formatted as `(my_user_id, movieID, rating)`. As in the original dataset, ratings should be between 1 and 5 (inclusive). If you have not seen at least 10 of these movies, you can increase the parameter passed to `take()` in the above cell until there are 10 movies that you have seen (or you can also guess what your rating would be for movies you have not seen).

```
# TODO: Replace <FILL IN> with appropriate code
from pyspark.sql import Row
my_user_id = 0

# Note that the movie IDs are the *last* number on each line. A common error
was to use the number of ratings as the movie ID.
my_rated_movies = [
      (my_user_id, 1193, 3.5),
      (my_user_id, 914, 2.5),
      (my_user_id, 2355, 4.2),
      (my_user_id, 1287, 3.7),
      (my_user_id, 594, 3.1),
      (my_user_id, 595, 2.6),
      (my_user_id, 2398, 1.7),
      (my_user_id, 1035, 4.0),
      (my_user_id, 2687, 5.0),
      (my_user_id, 3105, 4.7),
      (my_user_id, 1270, 2.5),
     # The format of each line is (my_user_id, movie ID, your rating)
     # For example, to give the movie "Star Wars: Episode IV - A New Hope
(1977)" a five rating, you would add the following line:
     #   (my_user_id, 260, 5),
]

my_ratings_df = sqlContext.createDataFrame(my_rated_movies,
['userId','movieId','rating'])
print 'My movie ratings:'
display(my_ratings_df.limit(10))
```

userId	moviel
0	1193
0	914

0	2355
0	1287
0	594
0	595
0	2398

(3b) Add Your Movies to Training Dataset

Now that you have ratings for yourself, you need to add your ratings to the `training` dataset so that the model you train will incorporate your preferences. Spark's unionAll() (http://spark.apache.org/docs/1.6.2/api/python/pyspark.sql.html#pyspark.sql.DataFram transformation combines two DataFrames; use `unionAll()` to create a new training dataset that includes your ratings and the data in the original training dataset.

```
# TODO: Replace <FILL IN> with appropriate code
training_with_my_ratings_df = training_df.unionAll(my_ratings_df)

print ('The training dataset now has %s more entries than the original training
dataset' %
       (training_with_my_ratings_df.count() - training_df.count()))
assert (training_with_my_ratings_df.count() - training_df.count()) ==
my_ratings_df.count()
```

```
The training dataset now has 11 more entries than the original training datase
t
```

(3c) Train a Model with Your Ratings

Now, train a model with your ratings added and the parameters you used in in part (2b) and (2c). Mke sure you include **all** of the parameters.

Note: This cell will take about 30 seconds to run.

```
# TODO: Replace <FILL IN> with appropriate code

# Reset the parameters for the ALS object.
als.setPredictionCol("prediction")\
   .setMaxIter(5)\
   .setSeed(seed)\
   .setRegParam(0.1)\

.setUserCol("userId").setItemCol("movieId").setRatingCol("rating").setRank(rank
s[best_rank])


# Create the model with these parameters.
my_ratings_model = als.fit(training_with_my_ratings_df)
```

(3d) Check RMSE for the New Model with Your Ratings

Compute the RMSE for this new model on the test set.

- Run your model (the one you just trained) against the test data set in `test_df` .
- Then, use our previously-computed `reg_eval` object to compute the RMSE of your ratings.

```
# TODO: Replace <FILL IN> with appropriate code
my_predict_df = my_ratings_model.transform(test_df)

# Remove NaN values from prediction (due to SPARK-14489)
predicted_test_my_ratings_df = my_predict_df.filter(my_predict_df.prediction !=
float('nan'))

# Run the previously created RMSE evaluator, reg_eval, on the
predicted_test_my_ratings_df DataFrame
test_RMSE_my_ratings = reg_eval.evaluate(predicted_test_my_ratings_df)
print('The model had a RMSE on the test set of
{0}'.format(test_RMSE_my_ratings))
```

```
The model had a RMSE on the test set of 0.81131748181
```

(3e) Predict Your Ratings

So far, we have only computed the error of the model. Next, let's predict what ratings you would give to the movies that you did not already provide ratings for.

The steps you should perform are:

- Filter out the movies you already rated manually. (Use the `my_rated_movie_ids` variable.) Put the results in a new `not_rated_df` .

  Hint: The Column.isin() (http://spark.apache.org/docs/1.6.2/api/python/pyspark.sql.html#pyspark.sql.Colum method, as well as the `~` ("not") DataFrame logical operator, may come in handy here. Here's an example of using `isin()` :

```
 > df1 = sqlContext.createDataFrame([("Jim", 10), ("Julie", 9), ("Abdul", 20), ("Mireill
e", 19)], ["name", "age"])
 > df1.show()
 +--------+---+
 |    name|age|
 +--------+---+
 |     Jim| 10|
 |   Julie|  9|
 |   Abdul| 20|
 |Mireille| 19|
 +--------+---+

 > names_to_delete = ["Julie", "Abdul"] # this is just a Python list
 > df2 = df1.filter(~ df1["name"].isin(names_to_delete)) # "NOT IN"
 > df2.show()
 +--------+---+
 |    name|age|
 +--------+---+
 |     Jim| 10|
 |Mireille| 19|
 +--------+---+
```

- Transform `not_rated_df` into `my_unrated_movies_df` by:
  - renaming the "ID" column to "movieId"
  - adding a "userId" column with the value contained in the `my_user_id` variable defined above.

- Create a `predicted_ratings_df` DataFrame by applying `my_ratings_model` to `my_unrated_movies_df` .

```
# TODO: Replace <FILL_IN> with the appropriate code

# Create a list of my rated movie IDs
my_rated_movie_ids = [x[1] for x in my_rated_movies]

# Filter out the movies I already rated.
not_rated_df = movies_df.filter(~ movies_df["ID"].isin(my_rated_movie_ids))

# Rename the "ID" column to be "movieId", and add a column with my_user_id as
"userId".
my_unrated_movies_df = not_rated_df.selectExpr("ID as
movieId").withColumn('userId', F.lit(my_user_id))

# Use my_rating_model to predict ratings for the movies that I did not manually
rate.
raw_predicted_ratings_df = my_ratings_model.transform(my_unrated_movies_df)

predicted_ratings_df =
raw_predicted_ratings_df.filter(raw_predicted_ratings_df['prediction'] !=
float('nan'))
```

(3f) Predict Your Ratings

We have our predicted ratings. Now we can print out the 25 movies with the highest predicted ratings.

The steps you should perform are:

- Join your `predicted_ratings_df` DataFrame with the `movie_names_with_avg_ratings_df` DataFrame to obtain the ratings counts for each movie.
- Sort the resulting DataFrame (`predicted_with_counts_df`) by predicted rating (highest ratings first), and remove any ratings with a count of 75 or less.
- Print the top 25 movies that remain.

```
My 25 highest rated movies as predicted (for movies with more than 75 review
s):
+-------+------+----------+-------+-----+-------------------+------------------
```

```
--+
|movieId|userId|prediction|movieId|count|           average|               tit
le|
+-------+------+----------+-------+-----+------------------+------------------
--+
|    318|     0|  4.208265|    318|63366| 4.446990499637029|Shawshank Redemp
t...|
|  92259|     0| 4.1027794|  92259| 2738| 4.132395909422937| Intouchables (201
1)|
| 108548|     0|  4.073626| 108548|   83|3.4216867469879517|Big Bang Theory,
 ...|
|    356|     0|  4.057464|    356|66172| 4.029000181345584| Forrest Gump (199
4)|
|   2324|     0|  4.037293|   2324|18156| 4.175837188808107|Life Is Beautifu
l...|
|   1704|     0| 3.9686944|   1704|28324| 4.032516593701454|Good Will Huntin
g...|
|  79132|     0| 3.9652848|  79132|14023| 4.156172003137702|    Inception (201
0)|
|    527|     0| 3.9622698|    527|50054| 4.310175010988133|Schindler's List
 ...|
|   3147|     0|  3.959468|   3147|21080|3.9826612903225804|Green Mile, The
 (...|
|   7153|     0| 3.9454288|   7153|31577|  4.14238211356367|Lord of the Ring
s...|
|   8132|     0| 3.9382305|   8132| 3595|3.9514603616133517|    Gladiator (199
2)|
|  72641|     0| 3.9380264|  72641| 2123|3.8657560056523788|Blind Side, The
 ...|
|   2571|     0| 3.9378889|   2571|51334| 4.187185880702848|  Matrix, The (199
9)|
|   3578|     0|  3.930058|   3578|32878| 3.952247703631608|    Gladiator (200
0)|
|   5952|     0|  3.909767|   5952|33947| 4.107520546734616|Lord of the Ring
s...|
|    110|     0| 3.9049845|    110|53769| 4.042533802004873|   Braveheart (199
5)|
|  88810|     0| 3.9022055|  88810| 1674|3.9166666666666665|    Help, The (201
1)|
|   4993|     0| 3.9016905|   4993|37553| 4.137925065906852|Lord of the Ring
s...|
|   4995|     0| 3.9010403|   4995|21931|  3.91974830149104|Beautiful Mind,
 A...|
|  31123|     0| 3.8989677|  31123|   76|3.6578947368421053|Ruby & Quentin
 (T...|
|  58559|     0| 3.8961282|  58559|20438| 4.220129171151776|Dark Knight, The
 ...|
|   6187|     0| 3.8916745|   6187| 1997| 3.627941912869304|Life of David Ga
l...|
|  72998|     0| 3.8895814|  72998| 9753|3.7765815646467753|       Avatar (200
9)|
|   8533|     0| 3.8891296|   8533| 3968|3.8040574596774195|Notebook, The (200
4)|
|     50|     0|   3.88542|     50|47006| 4.334372207803259|Usual Suspects,
 T...|
+-------+------+---------+-------+-----+------------------+------------------
--+
only showing top 25 rows
```