ISQA 8080

Homwork 6

Brian Detweiler

# Part 1. Questions for Spark

## Question 1: What time period does our dataset cover?

The ratingsArray is read in as Unicodes. In order to perform numerical min() and max() operations on it, we'll need to convert this column to ints.

>>> ratingsArray = ratings.map(lambda line: int(line.split(',')[3]))

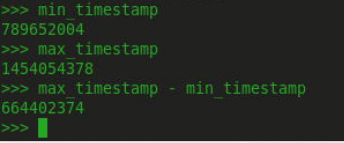
Then we can take the min() and max():

>>> min\_timestamp = ratingsArray.min()

>>> max\_timestamp = ratingsArray.max()

And lastly, we'll subtract them to get the timespan:

>>> max\_timestamp - min\_timestamp



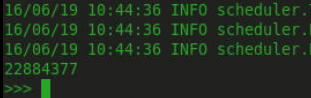
With these values, we can take them over to <http://www.epochconverter.com/> and find the actual GMT date/times.

The earliest timestamp is **GMT**: Mon, 09 Jan 1995 11:46:44 GMT

The latest timestamp is **GMT**: Fri, 29 Jan 2016 07:59:38 GMT

This time period spans 7689 days, 20 hours, 12 minutes and 54 seconds. Or, roughly 21 years. Crazy! If we count up the total number of reviews, we can get the average number of posts per day:

>>> ratingsArray.count()



Since the timestamp is in seconds, we divide by the number of seconds in a day, and then divide the total number of movie reviews by the timespan in days:



Over the course of 21 years, an average of 2976 movie reviews per day have been posted! Not bad!

## Question 2: What is the most watched movie?

For this question, we'll do a simple aggregate sum on the movie ID then sort by the top sum. Return them with the key as the movie ID, and the sum as the value.

>>> most\_ratings = ratings.map(lambda line: (line.split(',')[1], 1)).reduceByKey(lambda r1, r2: r1 + r2).map(lambda l: (l[1], l[0])).sortByKey(0).map(lambda l: (l[1], l[0]))

>>> most\_ratings.take(5)



Now, this isn't terribly helpful, because I don't know movies by ID numbers, I know them by title. So let's join the movies file and see what movies are watched the most.

>>> movies\_data = sc.textFile('/user/cloudera/movielens/input/movies.csv')

>>> header = movies\_data.first()

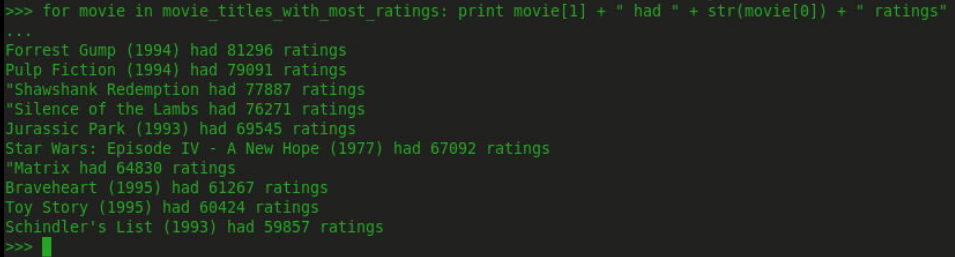
>>> movies = movies\_data.filter(lambda line: line != header)

>>> movie\_titles = movies.map(lambda l: (l[0], l[1]))

>>> movies\_with\_most\_ratings = most\_ratings.fullOuterJoin(movie\_titles)

>>> movie\_titles\_with\_most\_ratings = movies\_with\_most\_ratings.map(lambda l: l[1]).sortByKey(0).take(10)

>>> for movie in movie\_titles\_with\_most\_ratings: print movie[1] + " had " + str(movie[0]) + " ratings"



Forrest Gump is the most popular movie, followed by Pulp Fiction, Shawshank Redemption, etc. All good movies we'd expect to see in the top 10.

## Question 3: Who didn't like Pulp Fiction?

Pulp Fiction is a somewhat polarizing movie. Definitely not a movie for children, or the faint of heart. But it is the second most rated movie on our list. Let's get a list of everyone who hated pulp fiction by rating it 1.0 or less. First, we'll get the movie ID.

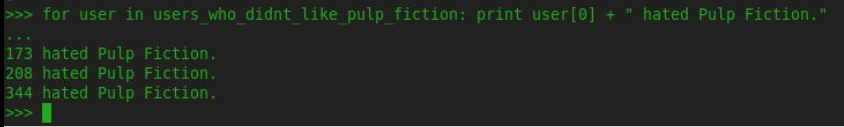
>>> movies\_data.map(lambda line: (line.split(',')[0], line.split(',')[1])).filter(lambda l: l[1] == "Pulp Fiction (1994)").take(1)



So we just need to filter out for both movie ID 296 and who rated it <= 1.0.

>>> users\_who\_didnt\_like\_pulp\_fiction = ratings.map(lambda l: (l.split(',')[0], l.split(',')[1], l.split(',')[2])).filter(lambda l: l[0] == u'296').map(lambda l: (l[1], l[2])).filter(lambda l: float(l[1]) <= 1.0).collect()

>>> for user in users\_who\_didnt\_like\_pulp\_fiction: print user[0] + " hated Pulp Fiction."



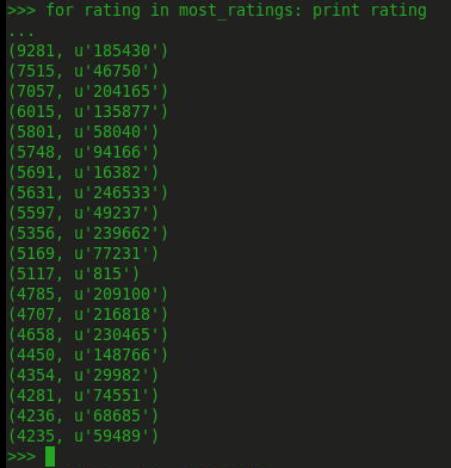
Surprisingly, only 3 people rated it 1 star!

# Part 2: Users with the most ratings

To get this, we'll need to use MapReduce. By mapping the line initially to tuples of (userId, 1), we can reduce on the userId as the key, and add up all the instances to get the total number of ratings per user. Then we need to map again, and reverse the result so that the key is the count. This allows us to sort on the count using sortByKey(0). We specify descending, as we only want the top 20.

>>> most\_ratings = ratings.map(lambda line: (line.split(',')[0], 1)).reduceByKey(lambda r1, r2: r1 + r2).map(lambda l: (l[1], l[0])).sortByKey(0).take(20)

>>> for rating in most\_ratings: print rating



User 185430 clearly has a lot of time on his/her hands!

# Part 3: Users that have as many ratings as my user ID number (7)

We've already halfway constructed the needed operations here. Now we just need to filter our results.

>>> ratings\_arr\_7 ratings.map(lambda line: (line.split(',')[0], 1)).reduceByKey(lambda r1, r2: r1 + r2).map(lambda l: (l[1], l[0])).filter(lambda l: l[0] == 7)

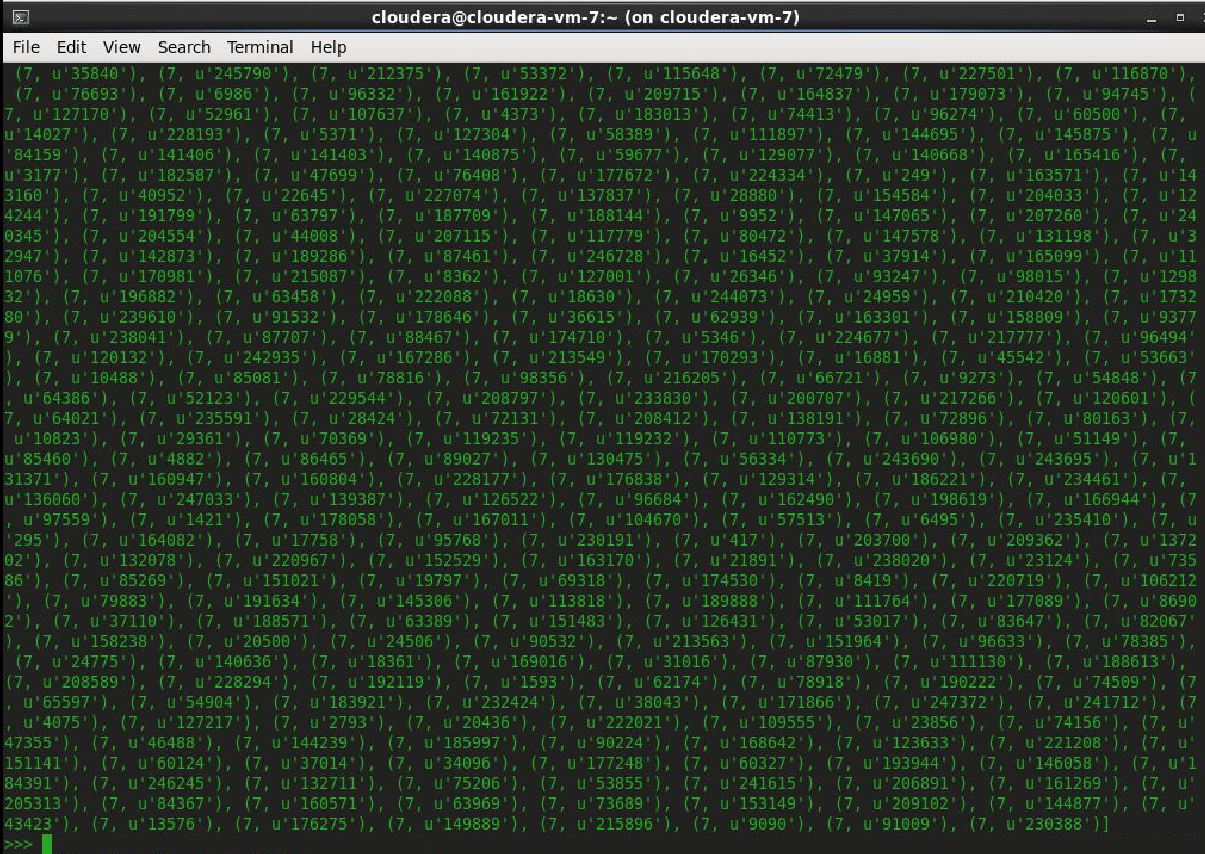
I have a feeling there are going to be lot of users with exactly 7 reviews.

>>> ratings\_arr\_7.count()



Yeah, that's a lot. But let's go ahead and get them.

>>> ratings\_arr\_7.collect()



It doesn't fit on one screen, but this gives you the idea.

# Bonus: Find the Average Ratings by User

I got a little help on this one from this tutorial:

<http://abshinn.github.io/python/apache-spark/2014/10/11/using-combinebykey-in-apache-spark/>

We'll start by mapping the ratings file to a user ID and rating tuple array. We need to take care to convert the strings to numerical values so we can work with them.

>>> sum\_count = ratings.map(lambda l: (int(l.split(',')[0]), float(l.split(',')[2])))

Then we use combineByKey() and pass it three lambdas. We'll describe these below.

>>> combined\_sum\_count = sum\_count.combineByKey(lambda value: (value, 1), lambda x, value: (x[0] + value, x[1] + 1), lambda x, y: (x[0] + y[0], x[1] + y[1]))

lambda value: (value, 1) - This creates a key-value tuple on which we can combine values.

lambda x, value: (x[0] + value, x[1] + 1): This is the merge value function that happens on each partition. It results in (key, (total, count)), where the key here is the userId, the total is the sum of all the ratings, and count is the number of ratings

lambda x, y: (x[0] + y[0], x[1] + y[1]): This is the merge combiners function, which merges all the partitions and gives us the final result, which looks like the previous step, but is combined across all partitions. It gives this result back to the driver.

Finally, we need to get the average, by dividing the value\_sum by the total count per userId.

>>> average\_by\_key = combined\_sum\_count.map(lambda (label, (value\_sum, count)): (label, value\_sum / count))

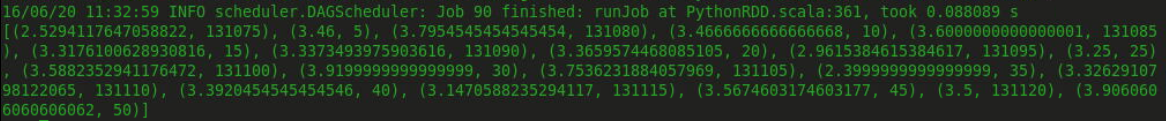
The mapper takes in the (label, (value\_sum, count)) tuple that we created in the previous step and simply returns a tuple with the average, which is obtained by dividing the value\_sum by the total count.

I then took this one step further and mapped the results to a (userId, average\_rating) tuple and ordered it by the highest 10 averages and lowest 10 averages.

>>> average\_by\_key.take(10)

>>> average\_by\_key.map(lambda l: (l[1], l[0])).sortByKey().take(10)

>>> average\_by\_key.map(lambda l: (l[1], l[0])).sortByKey().take(10)



Random ratings averages



Top 10 average ratings (all 5.0)



Bottom 10 average ratings (all 0.5)

As a side note, having a sample size and standard deviation along with the average would make this data much more useful, but that is outside the scope of this bonus problem.