SYNOPSIS AND DESCRIPTION:

RUSpark is a program that performs four types of data analysis operations on two types of data sets, RedditData and NetflixData. The four operations are:

- Analyze total image impact of a photo in Reddit
- Analyze most optimal time to post content on Reddit
- Analyze average movie ratings of Netflix movies
- Analyze Netflix movie ratings and create a recommendation graph

SUMMARY OF IMPLEMENTATION:

- Part I - Reddit Photo Impact:

  In order to extract the impact of each photo, I had to read in all lines of the CSV data and map each line to a Tuple2 of type <Integer, Integer> to store the key-value pairs of (imageID, sum(upvotes + downvotes + comments). Then I reduce by key to combine all posts that contain the same imageID and sum each corresponding sum to yield a final key-value pair of (imageID, totalSum(impact)). From there it is a simple collection of the pairs, sorting by key, then outputting the results.
- Part II - Reddit Hour Impact

  I created a SimpleDateFormat to extract only the hour form the unixtime in column 2 of the CSV file. From there, I inputted all lines, parsed the unixtime to extract the hour, computed the sum of the impact, and then mapped each line to a key-value pair of (hour, sum(impact)). Then, similar to photo impact, it was a matter of reducing by key, collecting the output, sorting it in ascending order of hour, and then outputting the results.
- Part III - Netflix Movie Average

  I read in all the lines, mapped each line to a key-value pair of type <Integer, Tuple2<Double, Integer>> to represent each line as it's movieID paired with a tuple that stores the rating and the count, e.g. (movieID, (rating, count)). The mapping sets the default count to 1. After this, we call reduceByKey to sum up all the ratings of each movie as well as the number of ratings of each movie. The resulting key-value pairs contain tuples (movieID, (sum(ratings), sum(counts)). We then map each of these to a new tuple to have (movieID, sum(ratings) / sum(counts)) by doing another mapToPair. Lastly, as before, we then collect, sort, and output the data.
- Part IV - Netflix Graph Generate

  We first read in all lines, parse each line to the relevant data types, then map each line's data to a pair to represent ((movieID, rating), custID). This now has a collection of tuples that contain each unique (movieID, rating) tuple along with each customer that gave said rating to said movie. Next, we perform a cartesian product to simulate a self-join on matching keys (said keys again being

(movie, rating) tuples). In the cartesian operation, we must filter on said matching keys and also filter to not double count each connection or count connections to itself. This was achieved by ensuring custID1 < custID2 for each new tuple created - said tuple is of type:

>> (((movieID1, rating1), custID1), ((movieID2, rating2), custID2))

>> movieID1 = movieID2, rating1 = rating2, and custID1 < custID2.

After that, we map this set of data to new pairs of tuples representing ((custID1, custID2), count), where each count is default 1. Next, we reduceByKey to sum up all counts between custID1 and custID2, collect the result, sort first by weight then by custID1 then by custID2, and output the result.

SECONDARY QUESTIONS:

- Based on the result from RedditPhotoImpact, what was the most impactful photo for the whole data set?
  - Answer: ImageID 1437 with total impact of 192896
- Based on the result from RedditHourImpact, what hours of the day (in EST) did submitted posts have the most impact/reach?
  - Answer: Hours 19-21 with impacts 11693660, 13898927, 14708461, respectively
- Based on the result from NetflixMovieAverage, name 2-3 movies that had the highest average rating.
  - Answer:
    - 14961 - *Lord of the Rings: The Return of the King: Extended Edition* with rating 4.72
    - 7230 - *The Lord of the Rings: The Fellowship of the Ring: Extended Edition* with rating 4.72
    - 7057 - *Lord of the Rings: The Two Towers: Extended Edition* with 4.70

ANALYSIS AND REFLECTION:

I think I took a reasonable approach in solving the problems that arose during the data analysis process. One notable exception might be in my handling of finding all unique pairs of customers who have identical movie-rating values for NetflixGraphGenerate. I intended to simulate a SQL join by doing the Cartesian product and then filtering the result, however this is likely computing a lot more pairs than necessary. Nonetheless, the only alternative I considered was groupByKey which I understand is very computationally expensive, hence why I tried to avoid it. I also could have used aggregateByKey, however I did not notice that until I already completed the current version and, given the performance wasn't too bad, I thought it was ok as is.

I think the hardest parts of the project were figuring out how Spark works and how to properly set up the map and reduce operations to extract the data we want. Spark does not have the best documentation, much of which is written for Scala, not Java, so it was not as easy as I

was hoping it would be to figure out what exactly it's doing with each method call. Moreover, it was not always clear what exactly should be done to the data to extract the values or to compute the desired end results. This was in particular the case for NetflixGraphGenerate, where I was not sure how best to extract all relevant pairs of customers given equal movie and rating. I further had some issues with the nesting of tuples and properly implementing the lambda expressions for filtering and for sorting using Comparators. Nonetheless, I was able to overcome these obstacles by exploring as much relevant Spark documentation and examples online as possible as well as reading more about how to properly manipulate and extract the data, particularly in the intermediary steps, to properly yield the desired resulting data.