DataSet URL: https://grouplens.org/datasets/movielens/

Context: GroupLens Research has collected and made available rating data sets using MovieLens web site (http://movielens.org). The data sets were collected over various periods of time and we will be using 20M dataset (190 MB). It includes 20 million ratings and 465,000 tag applications applied to 27,000 movies by 138,000 users, tag genome data with 12 million relevance scores across 1,100 tags. The attributes are user id, movie id, title, genre, rating, timestamp, tag, tagid, relevance, imdbid (used to find out IMDB ratings), tmdbid.

Dataset & Attributes information: available in http://files.grouplens.org/datasets/movielens/ml-20m-README.html

Problem Statement:

We want to solve different interesting things from this dataset such as

a. Grouping movies by genres (useful to find how many action movies are there for example),
b. How many movies were rate before and after 2000 and the average ratings differences between them.
c. We might also want to know the movies released by year.
d. We also want to know the movies with highest average ratings and
e. Highest number of ratings by user.
f. We should put a threshold on the number of ratings (for example: at least 500 ratings) when we are sorting by average rating.
g. Also, we might want to know the movies with long titles. The solutions were implemented using Spark RDD, Dataframe, joins, groupByKey, sort, map-reduce functions.

Set up environment: First a SparkContext object was created to access the GPS cluster and to do that, a SparkConf object was created with application information such as name (MovieRatings), spark executor memory etc.

```
val conf = new SparkConf()
  .setAppName("MovieRatings")
val sc = new SparkContext(conf)
```

Data Extraction:

We have 6 csv files containing movie information (id, title, genres), ratings information (user id, movie id, rating, when it was rated), tags and other files. First, we need to convert csv to rdd. MapPartitionsWithIndex was used to drop the row which contains the column names. It was used to drill down the columns of rdd and later join rdds. Each line was trimmed to avoid

unnecessary spaces. The home directory was set under tmp folder to reuse in conversion of all csv file to rdd. For example: movies.csv was converted top rdd like below:

```
val ratingsData = sc.textFile("/tmp/ml-20m/ratings.csv")
  .map(_.split(",")
    .map(elem => elem.trim))
  .mapPartitionsWithIndex { (idx, iter) => if (idx == 0) iter.drop(1) else iter }
  .map(x => Ratings(Integer.parseInt(x(0)), Integer.parseInt(x(1)), x(2).toFloat, x(3)))
```

Case class for Rating (We used case classes for other cases too as it makes the code more readable and usable for other developers):
Case class Ratings(userId: Integer, movieId: Integer, rating: Float, timestamp: String)

I also worked with dataframe to export from csv. In Spark 2 versions, it's really easy to use sqlContext with spark-csv package and then converting to rdd. Well defined schemas were used to save as dataframe.

```
val ratingsSchema = StructType(Array(StructField("userId", IntegerType, true),
  StructField("movieId", IntegerType, true),
  StructField("rating", FloatType, true),
  StructField("timestamp", StringType, true)))
```

```
//Load with ratingsSchema
val ratingsDFWithSchema = csvLoader.schema(ratingsSchema).load("/tmp/ml-
20m/ratings.csv")

//Use case class to convert to rdd
val ratingsDS = ratingsDFWithSchema.as[Ratings]
val ratingsRDDWithSchema = ratingsDS.rdd.cache()
```

Statistics:
Ratings: (ratingsRdd is converted from ratings dataframe)

```
val ratingStats = ratingsData.map(ratings => ratings.rating).stats()
```

count: 20000263,
mean: 3.525529,
stdev: 1.051989,
max: 5.000000,
min: 0.500000

Printing schema from dataframe:
//root
//|-- userId: string (nullable = true)

//|-- movieId: string (nullable = true)
//|-- rating: string (nullable = true)
//|-- timestamp: string (nullable = true)

Movie count:
val moviesCount = moviesRdd.count //20000263

Number of action movies:
val countsMoviesAction= moviesRdd
.filter(movie => movie.genres.toLowerCase.contains("action")).count()
println("Number of Action movies: " + countsMoviesAction) //2851

To cache the rdds in memory to avoid recomputation, we updated accordingly.
val moviesRdd = moviesDF.rdd.cache()

The genres were splitted by "|" when a movie was part of multiple genres. Swap was used to sort by count, flatMap to flatten the output , reduceByKey was used to sum up the count. The final genre distribution of movies:

```
val groupByGenres= moviesRdd.map(movie => movie.genres.trim.split("\\|"))
  .flatMap(x => x)
  .map(x => (x,1)).reduceByKey(_+_)
val sortedGroupByGenres = groupByGenres
  .map{case (k,v)=>(v,k)}
  .sortByKey(ascending = false).map(_.swap)
println("Genres count: " + groupByGenres.count) //20
println("Genres distribution of movies: " + sortedGroupByGenres.take(20).foreach(println))
```

Result:

```
Genres distribution of movies:
  (Drama,9952)
(Comedy,6545)
(Romance,3186)
(Thriller,3131)
(Action,2851)
(Crime,2251)
(Documentary,1979)
(Horror,1949)
(Adventure,1750)
(Sci-Fi,1394)
(Mystery,1108)
(Fantasy,1062)
(War,889)
(Children,861)
(Animation,829)
```

```
(Musical,813)
(Western,486)
((no genres listed),235)
(Film-Noir,232)
(IMAX,166)
```

So, we see the top 5 genres are Drame, Comedy, Romance, Thriller and Action.

Movie distribution by year:

The release year of the movie is encoded into the last 6 substring of the title inside bracket (for example: (2006)). I used try catch to ignore the ones which couldn't be converted to an year format (bad data). Then countByKey was used to find number of movies released for each year in descending order.

```
val groupByYear= moviesRdd.map(movie => movie.title.trim.takeRight(6).trim)
  .map(x => try {Integer.parseInt(x.slice(1,5))}catch {case e: Exception => None})
  .map(x => (x,1)).reduceByKey(_+_)
println("Number of years movies have been released: "+ groupByYear.count)  //Number of
years movies have been released: 121

val sortedgroupByYear = groupByYear
.map(_.swap)
.sortByKey(false)
.map(_.swap)
val countMovieByYearFreq = sortedgroupByYear.countByKey()
println("Sorted movie count by year in descending order (top 10):" +
sortedgroupByYear.take(10).foreach(println))
```

Result:

```
Sorted movie count by year in descending order (top 10):
(2009,895)
(2013,842)
(2012,836)
(2011,802)
(2010,783)
(2008,778)
(2007,728)
(2006,670)
(2014,651)
```

So, the most movies that were rated in the data set are from 2009, 2013 and 2012 (top 3)

Highest average rated movies/Most rated movies by user:

First I calculated the number of users who rated each movie from Ratings rdd and the sum of the ratings for each movie id separately. Then both rdds where joined to find the average and then the computed rdd was joined with movie rdd to add movie title.

```
val groupByMovieIdRdd = ratingsData
.map(rating => (rating.movieId, 1))
.reduceByKey(_+_)
val groupByMovieIdRatingsRdd = ratingsData
.map(rating => (rating.movieId, rating.rating))
.reduceByKey(_+_)
case class MoviesAvg(movieId: Integer, ratingCount: Integer, ratingAvg: Float)
val joinedRdd = groupByMovieIdRdd
  .join(groupByMovieIdRatingsRdd)
  .map(x => (x._1, x._2._1, x._2._2/x._2._1))
.map(x => MoviesAvg(x._1, x._2, x._3))
val movieNameWithAvgRdd = joinedRdd
  .map(movieAvg => (movieAvg.movieId, movieAvg))
  .join(moviesRdd.map(movie => (movie.movieId, movie)))
case class MoviesNamesWithAvg(movieId: Integer, ratingCount: Integer, ratingAvg: Float, title:
String)
val movieNameWithAvgRddFinal = movieNameWithAvgRdd
  .map(x => (x._1 , x._2._1.ratingCount, x._2._1.ratingAvg, x._2._2.title))
  .map(x => MoviesNamesWithAvg(x._1, x._2, x._3, x._4))
val movieNameWithAvgRddFinalDF = movieNameWithAvgRddFinal.toDF
```

Result:

Top 5 rated movies: It is not useful as all of them have only 1 rating.

```
//movieNameWithAvgRddFinalDF.sort(desc("ratingAvg")).show(5)
//+-------+-----------+---------+--------------------+
//|movieId|ratingCount|ratingAvg|               title|
//+-------+-----------+---------+--------------------+
//| 129189|          1|      5.0|The Sea That Thin...|
//| 109253|          1|      5.0|Argentina latente...|
//| 106517|          1|      5.0|De la servitude m...|
//| 117061|          1|      5.0|    The Green (2011)|
//  | 121029|          1|      5.0|No Distance Left ...|
//+-------+-----------+---------+--------------------+
```

Top 5 most rated by users movies:

We find out “Pulp fiction” is the most rated movie in this data set

```
movieNameWithAvgRddFinalDF.sort(desc("ratingCount")).show(5)
+-------+-----------+---------+--------------------+
|movieId|ratingCount|ratingAvg|               title|
+-------+-----------+---------+--------------------+
|   296|      67310| 4.174231| Pulp Fiction (1994)|
  |   356|      66172|4.0290003| Forrest Gump (1994)|
  |   318|      63366|4.4469905|"Shawshank Redemp...|
|   593|      63299|4.1770563|"Silence of the L...|
```

```
|   480|     59715|3.6647408|Jurassic Park (1993)|
 +-------+----------+---------+--------------------+
```

Filter by rating count of at least 1000:

We find “Shawshank Redemption” is the top rated movie with average rating of 4.44 out of 5.

```
//movieNameWithAvgRddFinalDF.where("ratingCount >=
1000").sort(desc("ratingAvg")).show(10)
//+-------+----------+---------+--------------------+
//|movieId|ratingCount|ratingAvg|               title|
//+-------+----------+---------+--------------------+
//|    318|     63366|4.4469905|"Shawshank Redemp...|
//|    858|     41355|4.3647323|         "Godfather|
//|     50|     47006| 4.334372|     "Usual Suspects|
//|    527|     50054| 4.310175|Schindler's List ...|
//|   1221|     27398|4.2756405| "Godfather: Part II|
//|   2019|     11611|4.2741795|Seven Samurai (Sh...|
//|    904|     17449|4.2713337|  Rear Window (1954)|
//  |   7502|      4305| 4.263182|Band of Brothers ...|
//|    912|     24349| 4.258327|   Casablanca (1942)|
//  |    922|      6525|4.2569346|Sunset Blvd. (a.k...|
//+-------+----------+---------+--------------------+
```

We can do this calculation in a lot more easier way with dataframe’s aggregation (count,avg) methods, sql like coding.

```
val groupByMovieId = ratingsDFWithSchema
.groupBy("movieId")
val movieIdWithCountAndAverageRatingsDF = groupByMovieId.agg(count("rating")
.alias("count"), avg("rating").alias("average"))
val movieNameWithCountAndAvgRatingDF =
movieIdWithCountAndAverageRatingsDF.join(moviesDFWithSchema, Seq("movieId"))
val movieNamesWithAvgRatings = movieNameWithCountAndAvgRatingDF.select("average",
"title", "count", "movieId")
```

Top rated movie genres:

```
val joinedMovieRatingsRdd = movieNameWithCountAndAvgRatingDF.rdd
val ratingsByGenres = joinedMovieRatingsRdd
  .map(row => (row.getDouble(2),row.getString(4).trim.split("\\|")))
  .flatMapValues( x => x).map(_.swap)
  .mapValues(value => (value, 1))
  .reduceByKey((x, y) => (x._1 + y._1, x._2 + y._2))
  .mapValues{ case (sum, count) => (1.0 * sum) / count }
val ratingRank = ratingsByGenres
  .map(_.swap)
```

```
  .sortByKey(false).map(_.swap).take(20)
println("Top ranked movie genres: " + ratingRank.take(20).foreach(println))
```

Result:

```
Top rated movie genres: ()
(Film-Noir,3.4441508392873557)
(Documentary,3.436664522206702)
(War,3.321137043175274)
(IMAX,3.2946704049125257)
(Drama,3.26203806595643)
(Romance,3.20574298736737)
(Musical,3.182132878495172)
(Animation,3.1770952125106824)
(Crime,3.1671412122461153)
(Mystery,3.1350738677584125)
(Fantasy,3.093324230790382)
(Western,3.075228316860353)
(Comedy,3.0748694649959853)
(Adventure,3.072407494188218)
(Thriller,3.016095167324711)
(Action,2.976876289408412)
(Children,2.955607569754043)
(Sci-Fi,2.8902152347854897)
((no genres listed),2.8037878787878787)
(Horror,2.696795765281372)
```

So, we find the top 5 rated genres are Film-Noir, Documentary, War, IMAX and Drama. Also we notice only 330 Film-Noir types were in the set so probably we should vote for “War” types.

Changes in the millennium:

```
val filterYear= joinedMovieRatingsRdd
  .filter(row =>
    try {
      Integer.parseInt(row.getString(3).trim.takeRight(6).trim.slice(1,5)) >= 2000
    }
    catch {
      case e: Exception => false
    })
val ratingsByGenresMillenial = filterYear
  .map(row => (row.getDouble(2),row.getString(4).trim.split("\\|")))
  .flatMapValues( x => x).map(_.swap)
  .mapValues(value => (value, 1))
  .reduceByKey((x, y) => (x._1 + y._1, x._2 + y._2))
  .mapValues{ case (sum, count) => (1.0 * sum) / count }
```

```
val ratingRankMillenial = ratingsByGenresMillenial
  .map(_.swap)
  .sortByKey(false)
  .map(_.swap).take(20)  //takeOrdered is expensive
println("Top ranked movie genres after 2000: " + ratingRankMillenial.take(20).foreach(println))
```

Result: We found a slight difference in the ratings. “Documentaery” is the highest average rated movie in years after 2000

```
(Documentary,3.4329343458254837)
(Film-Noir,3.38129339803795)
(IMAX,3.295137874960822)
(War,3.2855363789808365)
(Drama,3.229571298677652)
(Musical,3.2020611673724217)
(Romance,3.1727093201939454)
(Animation,3.168726259815414)
(Crime,3.138970816413449)
(Fantasy,3.0775789121523123)
(Mystery,3.0770227544570203)
(Comedy,3.0595996210645806)
(Adventure,3.050792212059241)
(Western,3.030729430642586)
((no genres listed),2.990708812260536)
(Action,2.9605324049314596)
(Sci-Fi,2.945092379511849)
(Thriller,2.9414556594294377)
(Children,2.8787874979435295)
(Horror,2.66231545715875)
```

Movies with length greater than 10 and containing nothing other than alphabets:

```
val filterMovieTitlesPattern = moviesRdd.filter(movie => movie.title.split(" ") (0).length >= 10
&& movie.title.split(" ") (0).matches("[A-Za-z]+"))
println("Movies count with title length > 10 and only contains letters: " +
filterMovieTitlesPattern.count)    //1163
```

Use case class to convert it to rdd:

```
case class Movies(movieId: Integer, title: String, genres: String)  val moviesDS =
moviesDFWithSchema.as[Movies]  val moviesRDDWithSchema = moviesDS.rdd
```

Result:

println(moviesRDDWithSchema.take(5).foreach(println))

Movies(1,Toy Story (1995),Adventure|Animation|Children|Comedy|Fantasy)

Movies(2,Jumanji (1995),Adventure|Children|Fantasy)

Movies(3,Grumpier Old Men (1995),Comedy|Romance)

Movies(4,Waiting to Exhale (1995),Comedy|Drama|Romance)
Movies(5,Father of the Bride Part II (1995),Comedy)

Converting to Parquet and use registertemptable & sqlcontext to query like Sql:

```
moviesDFWithSchema.write.parquet("movies.parquet")
 val moviesParquet = sqlContext.read.parquet("movies.parquet")
 moviesParquet.registerTempTable("movies")
 sqlContext.sql("SELECT * FROM movies WHERE title.length > 10 limit 10").show
```

We can also convert to avro data format.

Use accumulator: They are distributed counters that are aggregated and available to the driver and to the job status page. It is useful for error counting and other monitoring tasks.

```
val filterMovieTitlesCount = sc.accumulator(0)
//  Use accumulators
val filterMovieTitles = moviesRdd.map(movie =>  if(movie.title.split(" ") (0).length >= 10)
filterMovieTitlesCount +=1)
```

Suggestions to improve the code:

We should mopdularize the code in difference packages such as:

Processor: This package will have all the actions as methods such as getTopRatedMovies, getMostTimesRatedMovies etc. with proper parameters.

Runner: This package will have the main class which will call the processor methods to print out to the console.

Model: This package will contain all the case classes as objects.

Util: This package will have the reusable codes such as parsing Strings, Spark-csv connector, Spark-avro converter etc.

In this way the code will be more organized, less chance of making mistakes, easy for other engineers to use, easier to debug. We should also use Gradle which I find to be more customizable than Maven Pom files.

For time constraint, the code wasn't run as a spark-submit job, instead all the results were achieved using spark2-shell. But if we were running the Scala code as a sparkm-submit job, the process will be like below:

Build with gradle and/or sbt:

Sbt file:

MovieRatings.sbt:

```
name := "Movie Ratings Project"

version := "1.0"

scalaVersion := "2.11.6"
```

libraryDependencies += "org.apache.spark" %% "spark-core" % "2.3.0"

Spark-submit job:
The arguments are below:

Class name “MovieRatings” scala file
The jar file name “spark-project_2.11-1.0.jar”
Directory path of the csv files (“/tmp/ml-20m”)
The output file “MoviesOut”

Moving jar to cluster: scp ~/test/target/scala-2.11/spark-project_2.11-1.0.jar khan7912@hc.gps.stthomas.edu:/home/khan7912

```
hadoop fs -rm -r MoviesOut

spark-submit \ --class MovieRatings \

    --master yarn-cluster \

    /home/khan7912/spark-project_2.11-1.0.jar\

    /tmp/ml-20m/ \

    MoviesOut\. 1

hadoop fs -cat MoviesOut/part-00000
```