Movie Recommender using MapReduce

Summary

The program uses as input parsed data from online platforms, in the form of csv files and by providing a userId and movieId and returns a top 5 movie selection using the following criteria:

1. Gengres' relativity (movies with the highest matching in relation to the given movie appear on top)
2. Populatity and ratings (aggregate of all the ratings by users that have evaluated each movie)
3. Release year (presenting movies close to the release year of the given movie - in the same decade)

Below are all the steps of the process analysed and commented.

Process

As a first step I import all the necessary libraries that will be used.

In [2]:
```
import re

from pyspark.sql.types import StringType
from pyspark import SQLContext
sqlContext = SQLContext(sc)
# from pyspark.sql import functions as f
# import pyspark.sql.functions
```

I enter a test case for presentation purposes. The test movie is "Toy Story", an Adventure, Animation, Children, Comedy, Fantasy movie released in 1995.

In [3]:
```
given_userid  = 7
given_movieid = 1
given_rating  = 3.0
```

First I import the data (list of movies and list of ratings) as pyspark dataframes, in order to work on them and clear up the initial data.

In [4]:
```
movies = sqlContext.read.load('movies.csv', format='com.databricks.spark.
ratings = sqlContext.read.load('ratings.csv', format='com.databricks.spar

movies.show(5)
ratings.show(5)
```

```
+-------+--------------------+--------------------+
|movieId|               title|              genres|
+-------+--------------------+--------------------+
|      1|    Toy Story (1995)|Adventure|Animati...|
|      2|      Jumanji (1995)|Adventure|Childre...|
|      3|Grumpier Old Men ...|      Comedy|Romance|
|      4|Waiting to Exhale...|Comedy|Drama|Romance|
|      5|Father of the Bri...|              Comedy|
+-------+--------------------+--------------------+
only showing top 5 rows

+------+-------+------+----------+
|userId|movieId|rating| timestamp|
+------+-------+------+----------+
|     1|     31|   2.5|1260759144|
|     1|   1029|   3.0|1260759179|
|     1|   1061|   3.0|1260759182|
|     1|   1129|   2.0|1260759185|
|     1|   1172|   4.0|1260759205|
+------+-------+------+----------+
only showing top 5 rows
```

Based on the movieId of the test movie, I find its genres.

In [5]:
```
given_genres = movies[movies.movieId.isin(given_movieid)].collect()[0][2]
given_genres
```

Out[5]:
```
['Adventure', 'Animation', 'Children', 'Comedy', 'Fantasy']
```

After that, based on the title of the movie, I extract the release year, which I use to create a decade ranging from -5 to +5. I do this taking into account that the movie industry is rapidly changing and movies need to belong to the same era to be relevant to each other.

In [6]:
```
# function identifying the release year of the movie based on the title
def release_year(title):
    a = re.split(r'[()-]', title)
    for i in range(len(a)):
        if a[i].isdigit() is True:
            b = int(a[i])
    return b

# release year of the given movie
title = movies[movies.movieId.isin(given_movieid)].collect()[0][1]
given_release_year = release_year(title)

# relevant movie year search range
year_range = list(range(given_release_year-4, given_release_year+6))
relevant_years = [str(x) for x in year_range]
```

I use the relevant year range and filter all the movies that were not released within this period, taking into account that all relevant movies shall be of the same period, irrespective of their genres and rating.

In [7]:
```
y_movies = movies.where(movies.title.contains(relevant_years[0]) | \
                        movies.title.contains(relevant_years[1]) | \
                        movies.title.contains(relevant_years[2]) | \
                        movies.title.contains(relevant_years[3]) | \
                        movies.title.contains(relevant_years[4]) | \
                        movies.title.contains(relevant_years[5]) | \
                        movies.title.contains(relevant_years[6]) | \
                        movies.title.contains(relevant_years[7]) | \
                        movies.title.contains(relevant_years[8]) | \
                        movies.title.contains(relevant_years[9]) )

y_movies.describe().show()
```

```
+-------+-----------------+--------------------+------------------+
|summary|          movieId|               title|            genres|
+-------+-----------------+--------------------+------------------+
|  count|             2341|                2341|              2341|
|   mean| 5804.74284493806|                null|              null|
| stddev|16663.34228526481|                null|              null|
|    min|                1|"""Great Performa...|(no genres listed)|
|    max|           145307|     eXistenZ (1999)|           Western|
+-------+-----------------+--------------------+------------------+
```

After that, based on the users' ratings, I identify the movies that the user has already seen, since I wont be reccomending a movie that the user has already seen. It needs to be noted that in case the given test movie was not selected correctly (aka there is no rating created by the given user for this movie) the results will be including the given movie.

In [8]:
```
seen_movies = ratings.filter(ratings.userId == given_userid).select('movi
seen_movies.describe().show()
```

```
+-------+-----------------+
|summary|          movieId|
+-------+-----------------+
|  count|               88|
|   mean|775.0568181818181|
| stddev|461.0633974896535|
|    min|                1|
|    max|             1408|
+-------+-----------------+
```

Then I complete a left anti-join, filtering out the movies that the user has already seen from the list of potential movie suggestions.

In [9]:
```
yu_movies = y_movies.join(seen_movies, ["movieId"], "leftanti")
yu_movies.describe().show()
```

```
+-------+------------------+--------------------+------------------+
|summary|           movieId|               title|            genres|
+-------+------------------+--------------------+------------------+
|  count|              2293|                2293|              2293|
|   mean| 5916.094635848233|                null|              null|
| stddev|16818.861513469856|                null|              null|
|    min|                 2|"""Great Performa...|(no genres listed)|
|    max|            145307|     eXistenZ (1999)|           Western|
+-------+------------------+--------------------+------------------+
```

Combine the two tables keeping only the columns that will be used in the next steps, I create a table with the rating, title and genres of each movie. Each line of this table is a different rating for a movie, made by a user.

In [10]:
```
movies_n_ratings = ratings.join(movies, ratings.movieId == movies.movieId
m_n_r = movies_n_ratings.drop('userId', 'movieId', 'timestamp')
m_n_r = m_n_r.selectExpr('rating as rating', 'title as title', 'genres as
m_n_r.describe().show()
```

```
+-------+------------------+--------------------+------------------+
|summary|            rating|               title|             genre|
+-------+------------------+--------------------+------------------+
|  count|            100004|              100004|            100004|
|   mean| 3.543608255669773|                null|              null|
| stddev|1.0580641091070326|                null|              null|
|    min|               0.5|"""Great Performa...|(no genres listed)|
|    max|               5.0| İtirazım Var (2014)|           Western|
+-------+------------------+--------------------+------------------+
```

The final list of movies' ratings is filtered, removing the movies that the user has already seen and movies that are not included in the 10 year period.

In [11]:
```
list_of_movies = m_n_r.join(yu_movies, ["title"], "inner").drop('movieId'
list_of_movies.describe().show()
```

```
+-------+--------------------+------------------+------------------+
|summary|               title|            rating|             genre|
+-------+--------------------+------------------+------------------+
|  count|               37818|             37818|             37818|
|   mean|                null|3.4449336294886033|              null|
| stddev|                null|1.0852797062079558|              null|
|    min|"""Great Performa...|               0.5|(no genres listed)|
|    max|     eXistenZ (1999)|               5.0|           Western|
+-------+--------------------+------------------+------------------+
```

I define a function calculating the genre relevance for each movie. Comparing the genres of each movie to the given one, the function gives a unique number for each set, adding value for every common genre and removing for every different one.

In [12]:
```
# Function defining relevance between a movie's genres and the given movi
def genre_relevance(genre):
    common_g = set(given_genres)&set(genre)
    different_g = set(given_genres)^set(genre)
    return len(common_g)/len(given_genres) - len(different_g)/(len(given_
```

In [13]:
```
list_of_movies.take(3)
```

Out[13]:
```
[Row(title='Sleepers (1996)', rating=3.0, genre='Thriller'),
 Row(title="Dracula (Bram Stoker's Dracula) (1992)", rating=3.5, genre
='Fantasy|Horror|Romance|Thriller'),
 Row(title='Cape Fear (1991)', rating=2.0, genre='Thriller')]
```

Then I turn the list of movies into a RDD, calculating the genre relevance of each movie.

In [14]:
```
movies_rdd = list_of_movies.rdd.map(lambda x:((x[0],genre_relevance(x[2].
movies_rdd.take(3)
```

Out[14]:
```
[(('Sleepers (1996)', -1.2), 3.0),
 (("Dracula (Bram Stoker's Dracula) (1992)", -1.2), 3.5),
 (('Cape Fear (1991)', -1.2), 2.0)]
```

I aggregate all the ratings for each movie together. The aggregate rating has two functions, as it takes into account both popularity as well as rating, suggesting that a popular movie would be a good suggestion despite having a relevant low score.

In [15]:
```
movies_rdd2 = movies_rdd.reduceByKey(lambda x,y:x+y)
movies_rdd2.take(2)
```

Out[15]:
```
[(('Sleepers (1996)', -1.2), 117.0),
 (("Dracula (Bram Stoker's Dracula) (1992)", -1.2), 171.5)]
```

I flatten the list of movies into tuples of Title - Genres Relevance - Aggregate score in order to have them sorted.

In [16]:
```
flatMappedRDD = movies_rdd2.map(lambda x: (x[0][0], x[0][1], x[1]))
(flatMappedRDD.take(1))
```

Out[16]:
```
[('Sleepers (1996)', -1.2, 117.0)]
```

The sorting of the movies is completed in order of genres relevance and aggregate score.

In [17]:
```
result_list = flatMappedRDD.sortBy(lambda a: ( -a[1], -a[2]))
result_list.take(5)
```

Out[17]:
```
[('Toy Story 2 (1999)', 1.0, 480.5),
 ('Antz (1998)', 1.0, 173.5),
 ("Emperor's New Groove, The (2000)", 1.0, 112.0),
 ('Adventures of Rocky and Bullwinkle, The (2000)', 1.0, 13.0),
 ('Space Jam (1996)', 0.8, 134.0)]
```

And finally the top 5 movies are presented in order of relevance.

In [18]:
```
top5 = result_list.map(lambda x: x[0])
print(*top5.take(5), sep='\n')
```

```
Toy Story 2 (1999)
Antz (1998)
Emperor's New Groove, The (2000)
Adventures of Rocky and Bullwinkle, The (2000)
Space Jam (1996)
```

As we can see the recommended movies are highly relevant to the given test movie, of the same type and period, while the user has not seen any of the listed movies.

In [19]:
```
title
```

Out[19]:
```
'Toy Story (1995)'
```

In [ ]:
```

```