3116 – Lab Distributed Data Analytics – Group 2

Exercise Sheet 8

Yuvaraj Prem Kumar

303384, premyu@uni-hildsheim.de

Distributed Computing with Apache Spark

We are using the latest version of Spark (PySpark) so there are support for programming with RDDs and DataFrames.

RDDs and DataFrames:

- Immutable (cannot be changed/updated/altered) once created.
- Track lineage information to efficiently recompute lost data.
- Enable operations on collection of elements in parallel.

There are two types of operations here, transformations and actions. Apache Spark uses lazy evaluation as a performance optimization technique. In Lazy evaluation, a transformation is not applied immediately to an RDD. Spark records the transformations that have to be applied to an RDD. Spark maintains the record of which operation is being called, through Directed Acyclic Graphs (DAG). Once an Action is called, Spark executes all the transformations. In MapReduce, much time is wasted in minimizing the number of MapReduce passes. It happens by clubbing the operations together. While in Spark we do not create the single execution graph, rather we club many simple operations. By design Spark transformations are lazy, for performance. So, we must use an action (collect, etc.) in order to prevent the 'laziness' of a transformation.

For PySpark implementation, we initialize the Spark session with :

```
sc = SparkSession \
    .builder \
    .appName("Python Spark") \
    .getOrCreate()
#OR#
sc = SparkContext.getOrCreate()
```

For latest version of Spark, SparkSession already contains SparkContext, so we can call it explicitly. I use 'getOrCreate' because I code in PyCharm IDE instead of notebooks so I run multiple .py files at the same time. We can only have one session; hence we 'get' the current one if so.

RDDs is for low-level transformations and actions, and if we want low-level control over our datasets. Optimization and performance are less of a concern as compared to DataFrames. There is no computation cost with sc.parallelize(). The code is lazy-evaluated as described above.

Part 1: Apache Spark Basics

A. Basic Operations on RDDs

We have these two lists, which are converted to RDDs

```
a = ["spark", "rdd", "python", "context", "create", "class"]
b = ["operation", "apache", "scala", "lambda", "parallel", "partition"]
rddA = sc.parallelize(a)
rddB = sc.parallelize(b)
```

Then we define a lambda function to make it a key-value pair for each word tuple

```
distA = rddA.map(lambda word: (word, 'Rdd_A'))
distB = rddB.map(lambda word: (word, 'Rdd_B'))
```

1. Right outer join and full outer join

   We use the PySpark built-in functions as show to return the join operations value

```
ro_join = distA.rightOuterJoin(distB).collect()
print("RIGHT OUTER JOIN: \n", ro_join, "\n")

fo_join = distA.fullOuterJoin(distB).collect()
print("FULL OUTER JOIN: \n", fo_join)
```

```
RIGHT OUTER JOIN:
 [('parallel', (None, 'Rdd_B')), ('lambda', (None, 'Rdd_B')), ('scala', (None, 'Rdd_B')), ('operation', (None, 'Rdd_B')), ('apache', (None, 'Rdd_B')), ('partition', (None, 'Rdd_B'))]

FULL OUTER JOIN:
 [('python', ('Rdd_A', None)), ('spark', ('Rdd_A', None)), ('context', ('Rdd_A', None)), ('create', ('Rdd_A', None)), ('parallel', (None, 'Rdd_B')), ('lambda', (None, 'Rdd_B')), ('class', ('Rdd_A', None)), ('rdd', ('Rdd_A', None)), ('scala', (None, 'Rdd_B')), ('operation', (None, 'Rdd_B')), ('apache', (None, 'Rdd_B')), ('partition', (None, 'Rdd_B'))]
```

2. Count of ‘s’ character - MapReduce

   As this is for both RDD A & B, first we UNION both RDDs. The first lambda function is to make into a list, then we can apply the lambda map then lambda reduce function [1]. The last lambda function combines the results from all partitions.

```
S_count= rddC.flatMap(lambda i:list(i)).map(lambda i:i.count('s')).reduce(lambda i,j:i+j)
print("Number of times 's' appears: ", S_count)
```

```
Number of times 's' appears:  4
```

3. Count of ‘s’ character – Aggregate

   This is more straight-forward, we just call ‘aggregate’ function to count it. There is an accumulating function within each partition - in this case count the letter ‘s’ on each row (x) and add to the accumulated count “i”.

```
S_count2=rddC.aggregate(0, lambda i, x: i + x.count('s'), lambda i, j: i+j)
print("Number of times 's' appears: ", S_count2)
```

```
Number of times 's' appears:  4
```

B. Basic Operations on DataFrames

This whole part B is solved using a wide variety of in-built pyspark.sql functions. We load the .json file using “spark.read.json”

```
Original json dataset:

+-----------------+-----------------+----------+---------+------+----+
|           course|              dob|first_name|last_name|points|s_id|
+-----------------+-----------------+----------+---------+------+----+
|Humanities and Art|  October 14, 1983|      Alan|      Joe|    10|   1|
|  Computer Science|September 26, 1980|    Martin|  Genberg|    17|   2|
|    Graphic Design|     June 12, 1982|     Athur|   Watson|    16|   3|
|    Graphic Design|     April 5, 1987|  Anabelle|  Sanberg|    12|   4|
|        Psychology|  November 1, 1978|      Kira| Schommer|    11|   5|
|          Business|  17 February 1981| Christian|   Kiriam|    10|   6|
|  Machine Learning|    1 January 1984|   Barbara|  Ballard|    14|   7|
|     Deep Learning|  January 13, 1978|      John|     null|    10|   8|
|  Machine Learning|  26 December 1989|    Marcus|   Carson|    15|   9|
|           Physics|  30 December 1987|     Marta|   Brooks|    11|  10|
|    Data Analytics|     June 12, 1975|     Holly| Schwartz|    12|  11|
|  Computer Science|      July 2, 1985|     April|    Black|  null|  12|
|  Computer Science|     July 22, 1980|     Irene|  Bradley|    13|  13|
|        Psychology|   7 February 1986|      Mark|    Weber|    12|  14|
|       Informatics|      May 18, 1987|     Rosie|   Norman|     9|  15|
|          Business|   August 10, 1984|    Martin|   Steele|     7|  16|
|  Machine Learning|  16 December 1990|     Colin| Martinez|     9|  17|
|    Data Analytics|              null|   Bridget|    Twain|     6|  18|
|          Business|      7 March 1980|   Darlene|    Mills|    19|  19|
|    Data Analytics|      June 2, 1985|   Zachary|     null|    10|  20|
+-----------------+-----------------+----------+---------+------+----+
```

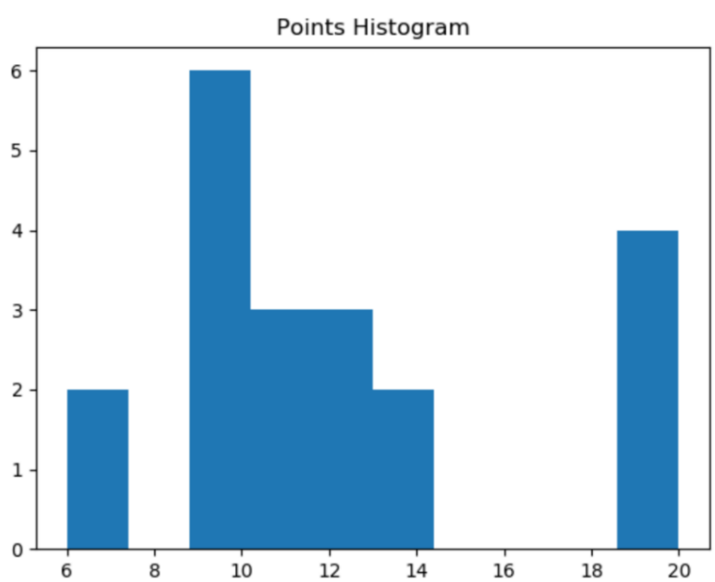
1. One liner, using df.na.fill(df.agg(mean())
2. One liner, using df.na.fill separately for the two columns.
3. We need several steps as there are several different formats in the original DOB column. Based on PySpark tutorial, the way is to convert from unix timestamp format to DD-MM-YYYY format. Hence, we should convert all the existing dates to unix timestamp format first, by creating a temp column.
    - 'MMMMM dd, yyyy' (October 14, 1983) to unix timestamp
    - ‘dd MMMMM yyyy' (30 December 1987) to unix timestamp
    - Change DOB column to the new temp column
    - Unix timestamp change to ‘dd-MM-yyyy’ with “to_date” function
    - Drop temp column
4. We can append a new column by “select *” then add the new column. It is the datediff of today’s datetime (system time) minus the DOB (converted). We need to use ‘round’ function otherwise the dataframe will give an error for the overflow. Then just name the new column with SQL ‘alias’.
5. We use the same df.agg function to calculate the mean of points, and add the df.agg(stddev). All using built-in functions. Then a simple if-else in pyspark (when points>calculated adjustment then 20, otherwise points).

6. Select points using pyspark select, with lambda function to select points for each row. We plot the points histograms using matplotlib.

Full code for 1-6 (references cited in .py file) and new output:

```
# 1. Replace the null value(s) in column points by the mean of all points.
df = df.na.fill(df.agg(mean('points')).collect()[0][0])

# 2. Replace the null value(s) in column dob and column last name by "unknown" and "--" respectively.
df = df.na.fill('unknown', 'dob')
df = df.na.fill('--', 'last_name')

# 3. Convert all the dates into DD-MM-YYYY format.
df2 = df.select('*', to_timestamp(df.dob, 'MMMMM dd, yyyy').alias('date'))
df2 = df2.withColumn('date', when(col('date').isNull(), to_timestamp(df.dob, 'dd MMMMM yyyy')).otherwise(col('date')))
df2 = df2.withColumn('dob', unix_timestamp(col("date"), 'yyyy-MM-dd HH:mm:ss').cast("timestamp"))
df2 = df2.withColumn('dob', to_date('dob', 'dd-MM-yyyy'))
df2 = df2.drop('date')

# 4. Insert a new column age and calculate the current age of all students.
df2 = df2.select('*', round(datediff(lit(datetime.date.today()), col('dob')) / 365, 0).alias('age'))

# 5. If point > 1 standard deviation of all points, then update current point to 20
grades = df2.agg({'points': 'mean'}).collect()[0][0] + df2.agg({'points': 'stddev'}).collect()[0][0]
df2 = df2.withColumn('points', when(col('points') > grades, 20).otherwise(col('points')))

print("Final json dataset: ", "\n")
df2.show()

# 6. Create a histogram on the new points created in the task 5.
points_histogram = df2.select('points').rdd.map(lambda i: i.points).collect()
plt.hist(points_histogram)
plt.title('Points Histogram')
plt.show()
```

```
Final json dataset:

+------------------+----------+----------+---------+------+----+----+
|            course|       dob|first_name|last_name|points|s_id| age|
+------------------+----------+----------+---------+------+----+----+
|Humanities and Art|1983-10-14|      Alan|      Joe|    10|   1|36.0|
|  Computer Science|1980-09-26|    Martin|  Genberg|    20|   2|39.0|
|    Graphic Design|1982-06-12|     Athur|   Watson|    20|   3|37.0|
|    Graphic Design|1987-04-05|  Anabelle|  Sanberg|    12|   4|32.0|
|        Psychology|1978-11-01|      Kira| Schommer|    11|   5|41.0|
|          Business|1981-02-17| Christian|   Kiriam|    10|   6|38.0|
|  Machine Learning|1984-01-01|   Barbara|  Ballard|    14|   7|36.0|
|     Deep Learning|1978-01-13|      John|       --|    10|   8|41.0|
|  Machine Learning|1989-12-26|    Marcus|   Carson|    20|   9|30.0|
|           Physics|1987-12-30|     Marta|   Brooks|    11|  10|32.0|
|    Data Analytics|1975-06-12|     Holly| Schwartz|    12|  11|44.0|
|  Computer Science|1985-07-02|     April|    Black|    11|  12|34.0|
|  Computer Science|1980-07-22|     Irene|  Bradley|    13|  13|39.0|
|        Psychology|1986-02-07|      Mark|    Weber|    12|  14|33.0|
|       Informatics|1987-05-18|     Rosie|   Norman|     9|  15|32.0|
|          Business|1984-08-10|    Martin|   Steele|     7|  16|35.0|
|  Machine Learning|1990-12-16|     Colin| Martinez|     9|  17|29.0|
|    Data Analytics|      null|   Bridget|    Twain|     6|  18|null|
|          Business|1980-03-07|   Darlene|    Mills|    20|  19|39.0|
|    Data Analytics|1985-06-02|   Zachary|       --|    10|  20|34.0|
+------------------+----------+----------+---------+------+----+----+
```



Part 2: Manipulating Recommender Dataset with Apache Spark

The movielens dataset is a rating prediction dataset with ratings given on a scale of 1 to 5. We are working with Tags Data File Structure "tags.dat", which contains data in the form {UserID::MovieID::Tag::Timestamp}. I follow the references from for implementation and coding [2].

Step 1: Data Pre-processing

We load the data using 'spark.read.csv' to get the original dataset as below:

```
Original tags dataset:

+---+----+----+----+---------------+----+----------+
|_c0| _c1| _c2| _c3|            _c4| _c5|       _c6|
+---+----+----+----+---------------+----+----------+
| 15|null|4973|null|     excellent!|null|1215184630|
| 20|null|1747|null|       politics|null|1188263867|
| 20|null|1747|null|         satire|null|1188263867|
| 20|null|2424|null|chick flick 212|null|1188263835|
| 20|null|2424|null|          hanks|null|1188263835|
| 20|null|2424|null|           ryan|null|1188263835|
| 20|null|2947|null|         action|null|1188263755|
| 20|null|2947|null|           bond|null|1188263756|
| 20|null|3033|null|          spoof|null|1188263880|
| 20|null|3033|null|      star wars|null|1188263880|
+---+----+----+----+---------------+----+----------+
only showing top 10 rows
```

- Drop NULL columns
- Rename columns as per dataset readme
- Select only distinct records (rows)
- Convert string timestamp to proper timestamp

We can obtain the pre-processed output as:

```
Modified tags dataframe:

+------+-------+-------------------+-------------------+
|UserID|MovieID|                Tag|          Timestamp|
+------+-------+-------------------+-------------------+
|    39|   2105|           computer|2007-08-28 03:17:00|
|   109|   7387|            zombies|2008-05-24 03:33:43|
|   146|      7|    based on a play|2008-11-15 10:52:44|
|   146|    351|               jazz|2007-10-07 11:54:37|
|   146|    364|    talking animals|2007-12-16 07:59:07|
|   146|    485|             parody|2006-10-12 13:33:20|
|   146|   1770|    based on a book|2007-09-25 12:24:21|
|   146|   2394|       Christianity|2008-03-04 06:55:58|
|   146|   3257|interracial romance|2008-02-25 09:22:17|
|   146|   7624|        prep school|2007-11-22 12:39:17|
+------+-------+-------------------+-------------------+
only showing top 10 rows
```

```python
df = spark.read.csv('C:/PythonProjects/tags.dat', sep=':')
print("Original tags dataset: ", "\n")
df.show(10)

df = df.drop('_c1', '_c3', '_c5')
df = df.selectExpr('_c0 as UserID', '_c2 as MovieID', '_c4 as Tag', '_c6 as Timestamp')
df = df.distinct()
df = df.withColumn('Timestamp', from_unixtime(df.Timestamp).cast(TimestampType()))

print("Modified tags dataframe: ", "\n")
df.show(10)
```

Step 2: Separate out tagging sessions for each user.

A tagging session for a user can be defined as the duration in which he/she generated tagging activities. Typically, an inactive duration of 30 mins is considered as a termination of the tagging session. We can use Rank function to identify the order of tagging per user. Once we ordered the dataset based on rank, we can calculate the time difference. This time difference is mapped to a new column ‘identifier’ which Boolean identifies which tags belong to which session (either 1 or 0). The way here is using PySpark “lag” or “window” functionality.

```python
w = Window.partitionBy(df['UserID']).orderBy(df['TimeStamp'].asc(), df['MovieID'].asc(), df['tag'].asc())
df = df.withColumn("Rank", dense_rank().over(w))

df.createOrReplaceTempView("tags")
df = spark.sql("SELECT a.*, b.Timestamp as Next_tag        \
                   FROM tags a                            \
                   left join tags b                       \
                    on a.UserID = b.UserID                \
                    and a.Rank = b.Rank - 1")


df = df.withColumn('Time_diff', unix_timestamp('Next_tag') - unix_timestamp('Timestamp'))
df = df.withColumn("Identifier", when(col('Time_diff') < 1800, 1).otherwise(0))
print("Tags dataframe with session identifier: ", "\n")
df.show(10)
```

```
Tags dataframe with session identifier:

+------+-------+--------------+-------------------+----+-------------------+---------+----------+
|UserID|MovieID|           Tag|          Timestamp|Rank|           Next_tag|Time_diff|Identifier|
+------+-------+--------------+-------------------+----+-------------------+---------+----------+
| 11563|  37830| final fantasy|2007-10-15 14:58:42|   1|               null|     null|         0|
|  1436|    838|       cottage|2007-09-02 18:44:43|   1|2007-09-02 18:45:02|       19|         1|
|  1436|   1953|action classic|2007-09-02 18:45:02|   2|2007-09-02 18:46:17|       75|         1|
|  1436|   1231|       awesome|2007-09-02 18:46:17|   3|2007-09-02 18:46:20|        3|         1|
|  1436|    247| coming of age|2007-09-02 18:46:20|   4|2007-09-02 18:46:34|       14|         1|
|  1436|   1994|           80s|2007-09-02 18:46:34|   5|2007-09-02 18:47:04|       30|         1|
|  1436|   1179|     new  nior|2007-09-02 18:47:04|   6|2007-09-02 18:47:26|       22|         1|
|  1436|  32587|    comic book|2007-09-02 18:47:26|   7|2007-09-02 18:47:48|       22|         1|
|  1436|   6378|        action|2007-09-02 18:47:48|   8|2007-09-02 18:47:57|        9|         1|
|  1436|   4447|        comedy|2007-09-02 18:47:57|   9|2007-09-02 18:48:28|       31|         1|
+------+-------+--------------+-------------------+----+-------------------+---------+----------+
only showing top 10 rows
```

Now we need to identify unique tagging session for each user. We cross join the dataframe (join the table to itself, using alias a,b). This is to duplicate the identifier column. It is used to identify the previous identifier tag. Now we can use a user-defined-function (UDF):

- If current identifier belongs to previous session, we assign back same session id.
- If not, we increment the session id by 1

```
def tagSession(identifier, prev_identifier):
    global cnt
    if prev_identifier is None:
        cnt = 1
        return cnt  # Initial record
    elif prev_identifier == 1:
        return cnt  # Return same session ID if same session
    elif prev_identifier == 0:
        cnt = cnt + 1
        return cnt  # Return increment session ID if new session
    else:
        return 0  # Unknown session


sessUDF = udf(tagSession)  # Initialize UDF
df = df.sort("UserID", "Rank")  # Sort (asc) on UserID, followed by Rank
df = df.withColumn('SessionID', sessUDF("identifier", "prev_identifier"))  # Append new column SessionID from UDF
df = df.drop('Next_tag', 'Identifier', 'prev_identifier', 'Time_diff')  # Drop the temp working columns

print("Modified Tags dataframe with each user session: ", "\n")
df.show(10)
```

```
Modified Tags dataframe with each user session:

+------+-------+----------------+-------------------+----+---------+
|UserID|MovieID|             Tag|          Timestamp|Rank|SessionID|
+------+-------+----------------+-------------------+----+---------+
|  1000|    277|children's story|2007-08-31 06:05:11|   1|        1|
|  1000|   1994|    sci-fi. dark|2007-08-31 06:05:36|   2|        1|
|  1000|   5377|         romance|2007-08-31 06:05:50|   3|        1|
|  1000|   7147|    family bonds|2007-08-31 06:06:01|   4|        1|
|  1000|    362|animated classic|2007-08-31 06:06:11|   5|        1|
|  1000|    276|          family|2007-08-31 06:07:15|   6|        1|
| 10003|  42013|        Passable|2006-06-16 06:33:55|   1|        1|
| 10003|  51662|  FIOS on demand|2008-04-12 00:35:26|   2|        2|
| 10003|  54997|  FIOS on demand|2008-04-12 00:35:35|   3|        2|
| 10003|  55765|  FIOS on demand|2008-04-12 00:35:42|   4|        2|
+------+-------+----------------+-------------------+----+---------+
only showing top 10 rows
```

Step 3: Frequency of tagging for each user session.

We select the required columns, and group by userID and new column SessionID. Sort output to visualise better.

```
df2 = df.groupby(['UserID', 'SessionID']).count() \
    .select('UserID', 'SessionID', col('count').alias('Freq_tags')) \
    .sort("UserID", "SessionID").cache()

print("Frequency tagging dataframe of UserID: ", "\n")
df2.show(10)
```

```
Frequency tagging dataframe of UserID:

[Stage 55:=============================
|UserID|SessionID|Freq_tags|
+------+---------+---------+
|  1000|        1|        6|
| 10003|        1|        1|
| 10003|        2|       18|
| 10003|        3|       38|
| 10020|        1|        2|
| 10025|        1|        1|
| 10032|        1|       39|
| 10032|       10|        1|
| 10032|       11|        1|
| 10032|       12|        1|
+------+---------+---------+
only showing top 10 rows
```

Step 4: Mean and standard deviation of the tagging frequency of each user

Using back the dataframe in Step 3, we call aggregate function to calculate the mean and standard deviation, then group by the UserID.

```
df3 = df2.groupby('UserID') \
    .agg(mean('Freq_tags').alias('Mean_tagging_freq')
         , count(lit(1)).alias('Sessions_count')
         , stddev('Freq_tags').alias('Std_Dev')) \
    .sort('UserID')

print("Mean and Standard deviation dataframe of UserID: ", "\n")
df3.show(10)
```

```
Mean and Standard deviation dataframe of UserID:

+------+------------------+--------------+------------------+
|UserID| Mean_tagging_freq|Sessions_count|           Std_Dev|
+------+------------------+--------------+------------------+
|  1000|               6.0|             1|               NaN|
| 10003|              19.0|             3|18.520259177452136|
| 10020|               2.0|             1|               NaN|
| 10025|               1.0|             1|               NaN|
| 10032| 4.666666666666667|            12|10.873933246182093|
| 10051|               1.0|             1|               NaN|
| 10058|25.333333333333332|             3|15.044378795195676|
| 10059|               2.5|             2|0.7071067811865476|
| 10064|               1.0|             1|               NaN|
| 10084|              3.75|             4|2.0615528128088303|
+------+------------------+--------------+------------------+
only showing top 10 rows
```

Step 5: Mean and standard deviation of the tagging frequency for across users.

Same as Step 4, but now we don't group by UserID, as we want to return the mean and standard deviation for all records in the dataframe.

```
df4 = df2.agg(mean('Freq_tags').alias('Mean_tagging_freq')
              , count(lit(1)).alias('Sessions_count')
              , stddev('Freq_tags').alias('Std_Dev'))

print("Total Mean and Standard deviation dataframe of UserID: ", "\n")
df4.show(10)
```

```
Total Mean and Standard deviation dataframe of UserID:

[Stage 77:==============================================>
|Mean_tagging_freq|Sessions_count|           Std_Dev|
+-----------------+--------------+------------------+
|8.270874880776901|         11533|30.563129861705935|
+-----------------+--------------+------------------+
```

Step 6: List of users with a mean tagging frequency within the two standard deviation from the mean frequency of all users.

First we get the mean and standard deviation from the dataframe in Step 5. Then we calculate the tags within 2 standard deviation from the mean. We filter only those records fulfiling this criteria (SQL where clause).

```
mean = df4.collect()[0]['Mean_tagging_freq']
std_dev = df4.collect()[0]['Std_Dev']

df5 = df3.filter((col('Mean_tagging_freq') >= (mean - (2 * std_dev))) &
                 (col('Mean_tagging_freq') <= (mean + (2 * std_dev)))) \
    .select('UserID')

print("List of users within 2 std dev from mean: ", "\n")
df5.show(10)
```

```
List of users within 2 std dev from mean:

+------+
|UserID|
+------+
|  1000|
| 10003|
| 10020|
| 10025|
| 10032|
| 10051|
| 10058|
| 10059|
| 10064|
| 10084|
+------+
```

Bonus

For this exercise we will use movielens10m dataset. The movielens10m dataset consists of 10 million rating entries by users. However since this is too large, I will use the 1m dataset. There are two main files required to solve this exercise 1) rating.dat and 2) movie.dat The rating.dat file contains userId, movieId, and ratings (on scale of 1 to 5). The movie.dat file contains information about movies i.e. movieId, Title and Genres. There are many examples for analysing this dataset, here I refer to [3] and [4].

Step 1: Data Pre-processing

The pre-processing steps here follow exactly as in Part 2. We obtain the processed dataset as follows:

```
Ratings dataframe:

+------+-------+------+-------------------+
|UserID|MovieID|Rating|          Timestamp|
+------+-------+------+-------------------+
|     5|    968|     3|2000-12-31 07:07:27|
|     5|    497|     3|2000-12-31 07:54:47|
|     6|   1210|     3|2000-12-31 05:16:59|
|     8|    377|     4|2000-12-31 03:20:04|
|    10|    745|     5|2001-01-11 00:16:04|
|    10|   1028|     5|2000-12-31 02:27:48|
|    10|   3252|     3|2000-12-31 03:12:07|
|    10|   1080|     5|2000-12-31 02:23:44|
|    12|    593|     5|2000-12-31 00:49:53|
|    13|   1372|     4|2000-12-30 19:44:44|
+------+-------+------+-------------------+
only showing top 10 rows
```

```
Movies dataframe:

]+-------+--------------------+--------------------+
 |MovieID|               Title|              Genres|
+-------+--------------------+--------------------+
|    432|   City Slickers II|                null|
|    762|   Striptease (1996)|        Comedy|Crime|
|   1073|Willy Wonka and t...|Adventure|Childre...|
|   1328|Amityville Curse,...|              Horror|
|   2060|  BASEketball (1998)|              Comedy|
|   2160|Rosemary's Baby (...|     Horror|Thriller|
|   2338|I Still Know What...|Horror|Mystery|Th...|
|   2354|Rugrats Movie, Th...|Animation|Childre...|
|   2580|           Go (1999)|               Crime|
|   2594|Open Your Eyes (A...|Drama|Romance|Sci-Fi|
+-------+--------------------+--------------------+
only showing top 10 rows
```

Step 2: Find the movie title which has the maximum average ratings?

We can inner join the two dataframes on movieID column, as we want the 'Title' from movie, and 'Rating' from rating. Then we get the mean, and group by movieID and append to column. Then apply filter on the column to get the aggregated maximum rating.

```
df_max = df_ratings.groupby('MovieID')                                    \
                    .agg(_mean_('Rating').alias('Max_avg_rating'))         \
                    .join(df_movie, df_ratings.MovieID == df_movie.MovieID)   \
                    .select(df_movie.MovieID,'Title', 'Max_avg_rating')
print("Movies with maximum avg rating: ", "\n")
df_max.filter(col('Max_avg_rating') == df_max.agg(_max_('Max_avg_rating')).collect()[0][0]).show(10)
```

```
Movies with maximum avg rating:

+-------+--------------------+--------------+
|MovieID|               Title|Max_avg_rating|
+-------+--------------------+--------------+
|   3656|        Lured (1947)|           5.0|
|   3280|     Baby, The (1973)|           5.0|
|    989|Schlafes Bruder (...|           5.0|
|    787|Gate of Heavenly ...|           5.0|
|   3607|One Little Indian...|           5.0|
|   3172|Ulysses (Ulisse) ...|           5.0|
|   3233|Smashing Time (1967)|           5.0|
|   3881|Bittersweet Motel...|           5.0|
|   1830|Follow the Bitch ...|           5.0|
|   3382|Song of Freedom (...|           5.0|
+-------+--------------------+--------------+
```

Step 3: User who has assign the lowest average ratings among all the users the number of ratings greater than 40.

With the same syntax as step 2, except here we use filter function for the total ratings >= 40. The total ratings are calculated from the mean and count. We use lit function to wrap to have acess to pyspark.sql.columns functions.

```
df_low = df_ratings.groupby('UserID').agg(_mean_('Rating').alias('Max_avg_rating'),_count_(lit(1)).alias('Total_ratings'))
                    .filter('Total_ratings >= 40')

lowest_rating = df_low.agg(_min_('Max_avg_rating'))
print("Users with with lowest rating assign: ", "\n")
df_low.filter(col('Max_avg_rating') == lowest_rating.collect()[0][0]).show(10)
```

```
Users with with lowest rating assign:

+------+-----------------+-------------+
|UserID|   Max_avg_rating|Total_ratings|
+------+-----------------+-------------+
|  3598|1.015384615384615З|           65|
+------+-----------------+-------------+
```

Step 4: Find the movie genre with the highest average ratings?

The readme provides the list of genre, we store as python list here. Then create a new dataframe by inner join ratings and movie on movieID column and only select genre. However this column will have contcatenated different genres. Hence we use explode outer to flatten it, and split out into individual genres. Then apply the same steps as previous, with mean and group by.

```
genre = ['Action', 'Adventure', 'Animation', 'Children', 'Comedy', 'Crime','Documentary', 'Drama', 'Fantasy',
         'Film-Noir', 'Horror', 'Musical','Mystery', 'Romance', 'Sci-Fi', 'Thriller', 'War', 'Western']

df_genre = df_ratings.join(df_movie, df_ratings.MovieID == df_movie.MovieID).select(df_movie.Genre, 'Rating')
print("Movie genre with highest avg rating: ", "\n")
df_genre.withColumn("Genre", explode_outer(split('Genre', "[|]")))          \
           .filter(col('Genre').isin(genre))                            \
           .groupBy('Genre')                                                \
           .agg(_mean_('Rating').alias('Max_rating'))                       \
           .orderBy(col('Max_rating').desc())                              \
           .show(1)
```

```
Movie genre with highest avg rating:

+---------+-----------------+
|   Genres|       Max_rating|
+---------+-----------------+
|Film-Noir|4.075187558184108|
+---------+-----------------+
only showing top 1 row
```

References:

[1] https://stackoverflow.com/questions/36559071/how-to-count-number-of-occurrences-by-using-pyspark
[2] https://medium.com/data-science-school/practical-apache-spark-in-10-minutes-part-3-dataframes-and-sql-ac36b26d28e5
[3] https://datascience-enthusiast.com/Python/cs110_lab2_als_prediction.html
[4] https://github.com/jesusgarcia2/MovieLens-Pyspark