TDT4305 - Project part 1

Henrik Kjærnsli & Patrik Kjærran - MTIØT
TDT4305 - Big Data Architecture

March 4, 2020

1 Introduction

Project part 1 revolves around analysis of the provided dataset, consisting of three *.csv* files. We have chosen to use Python's Spark API, *PySpark*, to solve the given tasks. Tasks 1 - 4 asks us to load the dataset into *resilient distributed datasets (RDDs)*, and use PySpark's builtin RDD functionalities to solve the exercises. Task 5 and 6 asks us to load the dataset into DataFrames, the equivalent of tables in relational databases, using Spark's SQL API to implement solutions. In the following sections we will describe our design and approach to the exercise as well as the results we obtained. We have added screen dumps of output for subtasks where output is reasonably small. All output files can be found in the *results* directory within the project root folder.

2 Deliverables

In addition to the report, the .zip file contains the source codes and *.csv* files with the output of our solutions. We have made one *.csv* file for each subtask, although task 5a and 6b are delivered as *.txt* files because of its format. Task 6a is by default exported as a cropped *.txt* due to its size (∼1.2 GB). See comment about changing its format at the top of the output file itself. Row delimiter is TAB for the output *.csv* files. Elements within a nested structure are separated by comma. See Appendix [5] for an overview of file output formats.

3 Setup

Install

This project requires Python 3.7 or above, along with a working installation of PySpark. For detailed instructions, see *spark_install.pdf* from course material on BlackBoard.

In terminal/command prompt, navigate to the project root directory and run:

```
pip install -r requirements.txt
```

Hadoop (Windows only): In order for file export to function correctly, precompiled Hadoop binaries is required (link).

Run

The script can be run from within the project root directory using

```
python -m tdt4305 [args ... ]
```

or

```
python main.py [args ... ]
```

Optional arguments:

`--action`: Denotes the action to perform for each task.

`run` (default): Prints results to console.

`export-csv`: Exports output as *.csv* files to project *output* directory.

`export-tsv`: Exports output as *.tsv* files to project *output* directory.

`export-txt`: Redirects console output to *output.txt* within project *output* directory.

`--tasks`: Specifies which tasks to execute (default: 1-6).

Example:

```
python -m tdt4305 --action export-csv --tasks 1,3,5-6
```

Input:

The script scans for the following files within the *data* directory:

- *yelp_businesses.csv*
- *yelp_top_reviewers_with_reviews.csv*
- *yelp_top_users_friendship_graph.csv*

Make sure to place the mentioned files into the data directory before running the code.

4 Tasks

Task 1

To solve the RDD-tasks, we load the data files into separate RDDs using `SparkContext.textFile(...)`.
We then remove the headers so it does not count toward any of the following results.
The function `rdd.count()` is applied to each RDD to count the number of rows.

```
--- TASK 1 ---
Business table size = 192609
Review table size = 883737
Friendship graph size = 1938472
```

Figure 1: Output from task 1.

Task 2

In task 2 we do quieries against the review-table to find our solutions.

a) We first map the RDD to only contain the user IDs. We then use the functions `rdd.distinct()` before `rdd.count()` to find the number of distinct users.

b) To find the average number of characters in a user review, we first map the RDD to only contain review lengths, before finding the average by summing the lengths and divide it by the number of reviews.

c) We find the business ID of the top 10 businesses with the most number of reviews by grouping the RDD on business ID. We then map values to the length by using `rdd.mapValues(len)` before a sorting is performed. Finally we map the RDD to only contain business ID and retreives the top 10 IDs from the sorted RDD by using `rdd.take(10)`.

d) By grouping the RDD on year, we can then use `rdd.mapValues(len)` to obtain reviews per year.

e) We first map RDD to only contain the review date. Because `datetime.fromtimestamp()` takes an integer as input, we need to convert the date value to an integer. We then obtain the first and last review date using `rdd.min()` and `rdd.max()`.

f) The goal in this subtask is to calculate Pearson Correlation Coefficient (PCC) between the number of reviews by a user and the average number of characters in the user's review.

   We first create key-value pairs of the form (*user_id*, *review_text*) for easier access to review texts, using `rdd.map(...)`.

   Next, we define three RDDs for later reference:

- **Number of reviews per user (rt_counts)**
  Obtained by grouping the RDD by key, before mapping the values to represent the number of reviews per user. Finally we use `rdd.values()` to only remain with the review count per user.

- **Total review length per user (rt_totals)**
  We first convert each review text in the mapping to its corresponding length using `rdd.mapValues(len)`. We map each user to its total review length using `rdd.reduceByKey(add)`, which groups by the first argument and aggregates by adding elements within the same group. At last, we convert this to a 1-dimensional RDD using `rdd.values()`.

- **Average review length per user (rt_counts)**
  Creating tuples of the form *(total_i, count_i)* using `rt_totals.zip(rt_counts)`. By mapping each tuple to the ratio between the first and second argument using `rdd.map(...)`, we get the average review length per user.

Looking at the provided formula, the PCC can now be divided into four parts:

- **Element-wise difference from average number of reviews (x_diff)**
  Average review length across all users is given by `rt_counts.sum() / rt_counts.count()`. Finally, we find the element-wise difference between the review count and the dataset average by subtracting the dataset average from each entry, using `rt_counts.map(...)`.

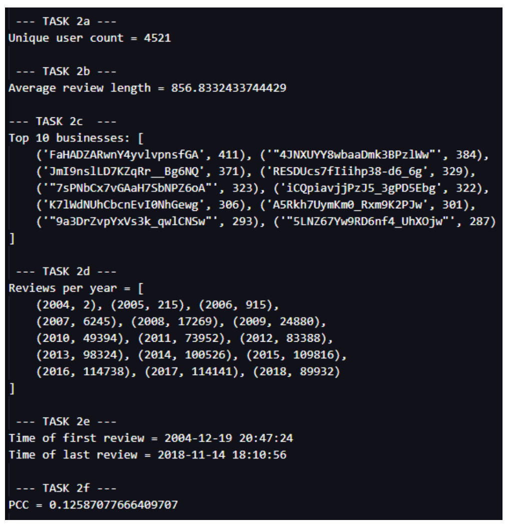
- **Element-wise difference from average review length (y_diff)**
  Average review length across all users is given by `rt_averages.sum() / rt_averages.count()`. Finally, we find the element-wise difference between the review lengths and the dataset average by subtracting the dataset average from each entry, using `rt_averages.map(...)`.

- **Sum of squared differences for number of reviews (sum_sqdiff_x)**
  We first find the element-wise square difference by squaring each element in x_diff using `rdd.map(...)`. The sum of squared review count differences is found by invoking `rdd.sum()` on the squared differences.

- **Sum of squared differences for review length (sum_sqdiff_y)**
  We first find the element-wise square difference by squaring each element in y_diff using `rdd.map(...)`. The sum of squared review length differences is found by invoking `rdd.sum()` on the squared differences.

Finally, we merge these four components together to form the PCC coefficient, as shown in the code snippet below.

```python
def task_2f(rt_rdd):
    """
    Calculates the 'Pearson correlation coefficient' between the number of reviews
    by a user and the average number of the characters in the user's reviews.
    """
    # Mapping of the form: 'user_id' -> 'decoded_review_text'
    rt_user_reviews = rt_rdd.map(lambda row: [row[1], row[3]])

    rt_counts = rt_user_reviews.groupByKey().mapValues(len).values()
    rt_totals = rt_user_reviews.mapValues(len).reduceByKey(add).values()
    rt_averages = rt_totals.zip(rt_counts).map(lambda row: row[0] / row[1])

    # Element-wise difference from average number of reviews
    X = rt_counts.sum() / rt_counts.count()
    x_diff = rt_counts.map(lambda x_i: x_i - X)

    # Element-wise difference from average review length
    Y = rt_averages.sum() / rt_averages.count()
    y_diff = rt_averages.map(lambda y_i: y_i - Y)

    # Sum of squared differences for number of reviews.
    sum_sqdiff_x = x_diff.map(lambda x_diff: x_diff**2).sum()

    # Sum of squared differences for review length.
    sum_sqdiff_y = y_diff.map(lambda y_diff: y_diff**2).sum()

    # Combine expressions
    numerator = x_diff.zip(y_diff).map(lambda diff: diff[0] * diff[1]).sum()
    denominator = sum_sqdiff_x**0.5 * sum_sqdiff_y**0.5
    return numerator / denominator
```



```
 --- TASK 2a ---
Unique user count = 4521

 --- TASK 2b ---
Average review length = 856.8332433744429

--- TASK 2c  ---
Top 10 businesses: [
    ('FaHADZARwnY4yvlvpnsfGA', 411), ('"4JNXUYY8wbaaDmk3BPzlWw"', 384),
    ('JmI9nslLD7KZqRr__Bg6NQ', 371), ('RESDUcs7fIiihp38-d6_6g', 329),
    ('"7sPNbCx7vGAaH7SbNPZ6oA"', 323), ('iCQpiavjjPzJ5_3gPD5Ebg', 322),
    ('K7lWdNUhCbcnEvI0NhGewg', 306), ('A5Rkh7UymKm0_Rxm9K2PJw', 301),
    ('"9a3DrZvpYxVs3k_qwlCNSw"', 293), ('"5LNZ67Yw9RD6nf4_UhXOjw"', 287)
]

 --- TASK 2d ---
Reviews per year = [
    (2004, 2), (2005, 215), (2006, 915),
    (2007, 6245), (2008, 17269), (2009, 24880),
    (2010, 49394), (2011, 73952), (2012, 83388),
    (2013, 98324), (2014, 100526), (2015, 109816),
    (2016, 114738), (2017, 114141), (2018, 89932)
]

 --- TASK 2e ---
Time of first review = 2004-12-19 20:47:24
Time of last review = 2018-11-14 18:10:56

 --- TASK 2f ---
PCC = 0.12587077666409707
```

Figure 2: Output from task 2.

Task 3

In task 3 we do quieries against the business-table to find our solutions.

a) We use the following approach to get the average rating for businesses in each city: Map the RDD so that it becomes on the form (city, (1, stars)). We then uses `rdd.reduceByKey(...)` to sum the stars for each city, as well as the ones in the tuple, before we divide the star count by the number of the review in the city.

b) The first part is to map the RDD so that it only contains categories. We then use `rdd.flatMap(...)` to create a row for each category-entry. Finally we use `rdd.countByValue()` to get the count for each review category, before using `rdd.top(10)` to extract the 10 most frequent entries.

c) To calculate the centroids, we need to sum the latitude and longitude for each postalcode and then divide it by the number of entries for the postalcode. We first map the RDD to have postal code as key, with latitude and longitude along with a counter variable as values. Afterwards, `rdd.reduceByKey(...)` is used to sum the counter variable, latitude and longitude separately for each postal code. To find the average centroid, we create tuples where we take the sum of the latitudes and divide it by the counter variable for each postal code. The same is done for the longitudes.

```
 --- TASK 3b ---
Category frequency
frequencies = [
    ('Breweries', 543), ('Chinese', 4428),
    ('Doctors', 5867), ('Food', 29989),
    ('Health & Medical', 17171), ('Mexican', 4618),
    ('Orthopedists', 306), ('Restaurants', 59371),
    ('Sports Medicine', 439), ('Weight Loss Centers', 792)
]
```

Figure 3: Output from task 3b.

Task 4

In task 4 we do quieries against the friendship-graph to find our solutions.

a) Highest degree source is found by first mapping the RDD to contain the source user IDs along with a counter variable. We then add the counter variables by using `rdd.reduceByKey(add)`, before using the `rdd.top(10)` function to return the top 10 nodes with the most out-degrees. Highest destination is found by exactly the same approach.

b) We start by finding the source degrees for each source user ID by using the same apporach as in 4*a*. The mean is then found by taking the mean of the values of the RDD by calling `rdd.mean()` on `rdd.values()`.

   To find the median, we first sort the RDD by the counter variable, before we merge the RDD by the use of `rdd.zipWithIndex()`. To be able to use `rdd.lookUp(...)`, we need to map the RDD so that the index is at index 1, while the counter variable is at index 0. We then perform a integer division to find the median. Exactly the same apporach is used to find the mean and median for the destination degrees.

```
 --- TASK 4a ---
Top ten input nodes = [
    ('"8DEyKVyplnOcSKx39vatbg"', 4919), ('ZIOCmdFaMIF56FR-nWr_2A', 4597),
    ('YttDgOC9AlM4HcAlDsbB2A', 4222), ('djxnI8Ux8ZYQJhiOQkrRhA', 4211),
    ('F_5_UNX-wrAFCXuAkBZRDw', 3943), ('dIIKEfOgo0KqUfGQvGikPg', 3651),
    ('GGTF7hnQi6D5W77_qiKlqg', 3609), ('NfU0zDaTMEQ4-X9dbQWd9A', 3557),
    ('"3gRfkaVcEWri-Ju7OQX7uQ"', 3396), ('NhgU7RhuYYFmpkb1jlYJ6Q', 3330)
]

Top ten output nodes = [
    ('ZIOCmdFaMIF56FR-nWr_2A', 9564), ('F_5_UNX-wrAFCXuAkBZRDw', 8586),
    ('djxnI8Ux8ZYQJhiOQkrRhA', 8381), ('YttDgOC9AlM4HcAlDsbB2A', 6758),
    ('NfU0zDaTMEQ4-X9dbQWd9A', 6506), ('dIIKEfOgo0KqUfGQvGikPg', 6187),
    ('ACUVZ4SiN0gni7dzVDm9EQ', 6065), ('"8DEyKVyplnOcSKx39vatbg"', 6026),
    ('w-w-k-QXosIKQ8HQVwU6IQ', 5987), ('Thc2zV-K-KLcvJn3fMPdqQ', 5821)
]

 --- TASK 4b ---
Mean input node degree = 12.060049149220657
Median input node degree = 1

Mean output node degree = 3.864161354240208
Median ouput node degree = 1
```

Figure 4: Output from task 4.

Task 5

To solve task 5, we specify the value types of each column, such that the types are in line with the ones given in the assignment. This is done using `sql.types.StructType` and `sql.types.StructField`.

```
 --- TASK 5a ---
(Business table schema)
root
 |-- business_id: string (nullable = true)
 |-- name: string (nullable = true)
 |-- address: string (nullable = true)
 |-- city: string (nullable = true)
 |-- state: string (nullable = true)
 |-- postal_code: string (nullable = true)
 |-- latitude: float (nullable = true)
 |-- longitude: float (nullable = true)
 |-- stars: float (nullable = true)
 |-- review_count: integer (nullable = true)
 |-- categories: string (nullable = true)

(Review table schema)
root
 |-- review_id: string (nullable = true)
 |-- user_id: string (nullable = true)
 |-- business_id: string (nullable = true)
 |-- review_text: string (nullable = true)
 |-- review_date: string (nullable = true)

(Friendship graph schema)
root
 |-- src_user_id: string (nullable = true)
 |-- dst_user_id: string (nullable = true)
```

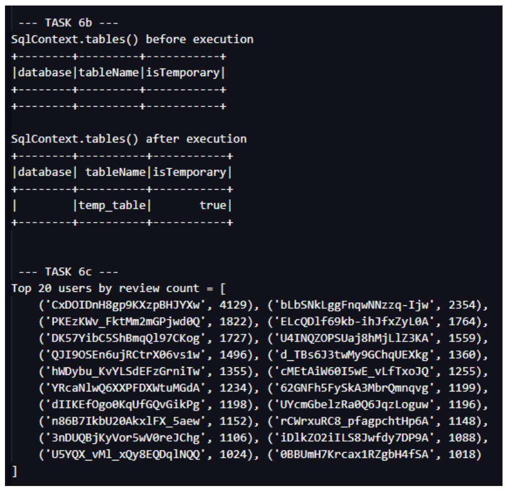
Figure 5: Output from task 5.

Task 6

a) To perform a inner join on the review table and business table, we use `rdd.join()`. The join is performed on business ID.

b) In task 6*b* we save the table from 6*a* in a temporary table. This is done by using `SqlContext.registerDataFrameAsTable()`.

c) We first group the DataFrame by user ID before sorting on the number of instances in descending order. Afterwards, we find the top 20 rows.



```
 --- TASK 6b ---
SqlContext.tables() before execution
+--------+---------+-----------+
|database|tableName|isTemporary|
+--------+---------+-----------+
+--------+---------+-----------+

SqlContext.tables() after execution
+--------+----------+-----------+
|database| tableName|isTemporary|
+--------+----------+-----------+
|        |temp_table|       true|
+--------+----------+-----------+


 --- TASK 6c ---
Top 20 users by review count = [
    ('CxDOIDnH8gp9KXzpBHJYXw', 4129), ('bLbSNkLggFnqwNNzzq-Ijw', 2354),
    ('PKEzKWv_FktMm2mGPjwd0Q', 1822), ('ELcQDlf69kb-ihJfxZyL0A', 1764),
    ('DK57YibC5ShBmqQl97CKog', 1727), ('U4INQZOPSUaj8hMjLlZ3KA', 1559),
    ('QJI9OSEn6ujRCtrX06vs1w', 1496), ('d_TBs6J3twMy9GChqUEXkg', 1360),
    ('hWDybu_KvYLSdEFzGrniTw', 1355), ('cMEtAiW60I5wE_vLfTxoJQ', 1255),
    ('YRcaNlwQ6XXPFDXWtuMGdA', 1234), ('62GNFh5FySkA3MbrQmnqvg', 1199),
    ('dIIKEfOgo0KqUfGQvGikPg', 1198), ('UYcmGbelzRa0Q6JqzLoguw', 1196),
    ('n86B7IkbU20AkxlFX_5aew', 1152), ('rCWrxuRC8_pfagpchtHp6A', 1148),
    ('3nDUQBjKyVor5wV0reJChg', 1106), ('iDlkZO2iILS8Jwfdy7DP9A', 1088),
    ('U5YQX_vMl_xQy8EQDqlNQQ', 1024), ('0BBUmH7Krcax1RZgbH4fSA', 1018)
]
```

Figure 6: Output from task 6b and 6c.

5 Appendix

Table 1: Output format for our solutions.

Task		Output file	Format
1	a	task_1a.csv	*table_name*, *table_size*
2	a	task_2a.csv	*unique_user_count*
	b	task_2b.csv	*avg_review_length*
	c	task_2c.csv	*business_id*, *review_count*
	d	task_2d.csv	*year*, *review_count*
	e	task_2e.csv	*datetime*
	f	task_2f.csv	*pcc*
3	a	task_3a.csv	*city*, *rating*
	b	task_3b.csv	*category*, *frequency*
	c	task_3c.csv	*postal_code*, *coordinates*
4	a	task_4a.csv	*(src_id_1, degree_1), (src_id_2, degree_2), ...* *(dest_id_1, degree_1), (dest_id_2, degree_2), ...*
	b	task_4b.csv	*src_mean, dest_mean* *src_median, dest_median*
5	a	task_5a.txt	—
6	a	task_6a.csv / .txt	*city, rating* / —
	b	task_6b.txt	—
	c	task_6c.csv	*user_id*, *review_count*