TDT4305 Big Data

Johan N. Slettevold & Jørgen S. Notland

Table of contents

Introduction

This report explains how the group has performed analysis on a large foursquare dataset. The analysis is performed with Apache Spark framework[1] using the pyspark python implementation.

1. Foursquare dataset

In all of the jobs written the group uses the following code to calculate how much time each job uses.

```
time0 = datetime.now()
 <THE CODE OF THE JOB>
time1 = datetime.now()
print "\nData init", time1-time0, "\n"
```

1.1 Load the Foursquare dataset.

Here we used the following methods to read the text files with the foursquare data and the city data into the program. sc.textFile returns the data in RDD format.

```
cities_data = sc.textFile("../foursquare-data/dataset_TIST2015_Cities.txt",
                    use_unicode=False)
foursquare_data = sc.textFile("../foursquare-data/dataset_TIST2015.tsv",
                    use_unicode=False)
```

The foursquare data has a header. Therefore we use the following methods to read the line with the header using .first(), and then filtering the header out of the dataset using the .filter() on the foursquare data. We remove the header because we want the dataset to be easier to work with.

```
foursquare_data_header = foursquare_data.first()
foursquare_data = foursquare_data.filter(lambda x: x !=
foursquare_data_header)
```

Here we map the cities_data and the foursquare_data RDD's with init functions. The init functions format the data and sets the data types of various fields on the data. This is to make the data easier to work with.

```
cities_data = cities_data.map(c_init)
foursquare_data = foursquare_data.map(f_init)
```

[1] http://spark.apache.org/

1.2 Calculate local time for each check-in (UTC time + timezone offset).

Use map on the foursquare_data to iterate the dataset.

```
foursquare_data_time = foursquare_data.map(set_time)
```

The set_time function takes the current time and adds the timezone offset to the time of the check in. The timedelta library is used to add the offset to the current time, and the datetime library is used to build a new time object. The corrected time for the checkin is set to the same column as the old time for the check in and the timezone offset is set to zero.

```
from datetime import datetime
from datetime import timedelta

def set_time(data):
      new_date = datetime.strftime(data[3] + timedelta(minutes=data[4]),
                  "%Y-%m-%d %H:%M:%S")
      data[3] = new_date
      data[4] = "0"
      return data
```

1.3 Assign a city and country to each check-in (use Haversine formula described below).

Here we iterate the foursquare data with the time columns added using a map function. The map function iterates all the cities in the city collection for every row in the foursquare data using a for loop and finds the closest of all the cities using the haversine formula. The closest city and country is appended to the row

```
cities_collection = cities_data.collect()

foursquare_data_locations = foursquare_data_time.map(assign_location)


def assign_location(foursquare_data):
      distance = sys.maxint
      city = None
      country = None
      for city_line in cities_collection:
```

```
      temp_distance = haversine(foursquare_data[5],
foursquare_data[6],
                        city_line[1], city_line[2])
if (temp_distance < distance):
      distance = temp_distance
      city = city_line[0]
      country = city_line[3]
foursquare_data.append(country)
foursquare_data.append(city)
return foursquare_data
```

1.4 Answer the following questions:

(a) How many unique users are represented in the dataset?

First the foursquare checkin data is mapped to a key value pair in the following format: (<user_id>, 1). Then we reduce the key_value RDD to only contain one of each key using the .reduceByKey() method. This is so that we get only one of each user ID in the RDD. Finally we use .count() to count the number entities in the RDD and find the number of unique users represented in the dataset.

```
key_value = foursquare_data.map(lambda x: (x[1], 1))
users = key_value.reduceByKey(add)
users_count = users.count()
```

Answer: 256307 unique users

(b) How many times did they check-in in total?

Here we use the .count() method to count the number of rows in the unmodified foursquare dataset. This returns the number of check-ins because the dataset consists of unique check-ins.

```
check_ins = foursquare_data.count()
```

Answer: 19265256 total check-ins

(c) How many check-in sessions are there in the dataset?

Here we use the same procedure as in a). The only difference is that Instead of using the user_id we map the tuples in the following format: (<session_id>, 1).

Then we reduce by key and count the number of entities with count() returning the total number of sessions.

```
key_value_sessions = foursquare_data.map(lambda x: (x[2], 1))
sessions = key_value_sessions.reduceByKey(add)
sessions_count = sessions.count()
```

Answer: 6338302 check-in sessions

(d) How many countries are represented in the dataset?

Here we use the same procedure as in a) and c). The only difference is that Instead of using the user_id we map the tuples in the following format: (<country>, 1). Then we reduce by key and count the number of entities with count() returning the total number of sessions.

```
key_value = foursquare_data_locations.map(lambda x: (x[9], 1))
countries = key_value.reduceByKey(add)
countries_count = countries.count()
time7 = datetime.now()
```

Answer: 6338302 check-in sessions

(e) How many cities are represented in the dataset?

Here we use the same procedure as in a), c) and d). The only difference is that Instead of using the user_id we map the tuples in the following format: (<city>, 1). Then we reduce by key and count the number of entities with count() returning the total number of cities.

```
key_value = foursquare_data_locations.map(lambda x: (x[10], 1))
cities = key_value.reduceByKey(add)
cities_count = cities.count()
```

Flaws: This code does not take into calculation that there could be multiple cities with the same name in the same or a different country. This flaw could have been corrected by using both city and country as key (lambda x: ((x[9], x[10]), 1)

1.5 Calculate lengths of sessions as number of check-ins and provide a histogram of these lengths.

First we filter out the checkin-in sessions from task 4c containing more than four check-ins. Then we map the data to the following format: (<Check-ins in session>, 1). We reduce this data by key so that we get the histogram with (<check-ins in session>, <frequency>).

```
filter_sessions = sessions.filter(lambda x: x[1]>4)
filter_sessions_tot = filter_sessions.map(lambda x: (x[1], 1))
filter_sessions_tot = filter_sessions_tot.reduceByKey(add)
```

Finally we get the following (unsorted) result on the dataset:

[(58, 12), (116, 1), (59, 15), (117, 3), (60, 9), (61, 15), (119, 2), (120, 1), (62, 9), (121, 1), (179, 1), (5, 295700), (63, 9), (64, 16), (354, 1), (180, 1), (6, 163683), (65, 6), (7, 96634), (8, 60479), (66, 7), (9, 40076), (67, 5), (125, 2), (10, 27245), (68, 11), (126, 1), (11, 19316), (69, 5), (128, 1), (12, 14106), (70, 10), (129, 1), (13, 10687), (71, 10), (72, 9), (14, 8001), (73, 4), (15, 6245), (16, 4854), (74, 4), (132, 1), (17, 3902), (75, 7), (18, 3200), (76, 2), (134, 1), (19, 2638), (77, 9), (135, 1), (136, 1), (20, 2125), (78, 4), (195, 1), (21, 1710), (79, 2), (22, 1487), (81, 3), (139, 1), (23, 1224), (24, 1075), (82, 3), (140, 1), (25, 898), (83, 3), (26, 751), (84, 2), (27, 587), (85, 4), (144, 1), (28, 505), (86, 1), (145, 1), (29, 417), (87, 2), (88, 1), (146, 1), (30, 381), (89, 1), (147, 2), (31, 301), (32, 264), (148, 1), (33, 238), (91, 2), (34, 179), (35, 181), (93, 2), (152, 2), (36, 144), (153, 1), (37, 150), (96, 2), (38, 130), (97, 1), (39, 115), (40, 85), (98, 2), (41, 99), (42, 79), (100, 3), (43, 70), (44, 77), (102, 2), (45, 55), (103, 1), (46, 57), (105, 1), (47, 48), (48, 50), (49, 41), (50, 33), (108, 3), (51, 32), (52, 28), (110, 1), (53, 17), (111, 1), (112, 1), (54, 22), (55, 17), (56, 14), (114, 1), (57, 20)]

1.6 For sessions with 4 and more check-ins, calculate their distance in kilometers (use Haversine formula to compute distance between two pairs of geo. coordinates).

In this task we first mapped the foursquare set to a tuple having the session id as key. Further on we had to use subtractByKey() twice to remove all sessions with less than four checkins. Then we grouped every checkins' time, lat, and lon to its respective session id. Finally we mapped this rdd with the haversine formula to calculate the distance between checkins.

```
key_value_sessions = foursquare_data_locations.map(lambda x:
```

```
                                                    (x[2], (x[3], x[5], x[6])))
filter_sessions_invert = sessions.subtractByKey(filter_sessions)
key_value_sessions =
key_value_sessions.subtractByKey(filter_sessions_invert)
key_value_sessions = key_value_sessions.groupByKey().mapValues(list)
key_value_sessions = key_value_sessions.map(set_distance)
time10 = datetime.now()

def set_distance(data):
      distance = 0
      for checkin in range(len(data[1])-1):
            distance += haversine(data[1][checkin][1], data[1][checkin][2],
                  data[1][checkin+1][1], data[1][checkin+1][2])
      return (data[0], distance)
```

1.7 Find 100 longest sessions (in terms of check-in counts) that cover distance of at least 50Km.

To find these 100 longest sessions, we decided to use the top() function; This design forced us to find the sessions covering 50km first. To do this we used a filter() and a subtractByKey() on filter_sessions from task 5. We then mapped this rdd so that the length of the session became the key, before finally using top(100).

```
key_value_sessions_invert = key_value_sessions.filter(lambda x: x[1]<50)
filter_sessions = filter_sessions.subtractByKey(key_value_sessions_invert)
key_value_checkins = filter_sessions.map(lambda x: (x[1], x[0]))
key_value_checkins = key_value_checkins.top(100)
time11 = datetime.now()
```

(a) For these 100 sessions, output data about their check-ins into a CSV or TSV file. Use all available data fields such as checkin_id, session_id, etc. and also add check-in date in `YYYY-MM-DD HH:MM:SS' format.

In this subtask we had to create two key value pair with session id’s as keys. This way we could find out which sessions in the original dataset we should include. Then we mapped the correct sessions with set_data, which formats the data nicely back into a .tsv format. We add a header before saving the output to disk.

```
key_value_checkins = sc.parallelize(key_value_checkins)
key_value_checkins = key_value_checkins.map(lambda x: (x[1], x[0]))
key_value_checkin_id = foursquare_data.map(lambda x: (x[2], (x[0],
x[1], x[3], x[5], x[6], x[7], x[8])))
key_value_checkin_id_invert =
key_value_checkin_id.subtractByKey(key_value_checkins)
key_value_checkin_id =
key_value_checkin_id.subtractByKey(key_value_checkin_id_invert)
foursquare_result = key_value_checkin_id.map(set_data)
foursquare_data_header =
sc.parallelize(["checkin_id\tuser_id\tsession_id\tzulu_time\tlat\tlon
\tcategory\tsubcategory"])
foursquare_result = foursquare_data_header.union(foursquare_result)
foursquare_result.saveAsTextFile("task7_results.tsv")

def set_data(data):
      result = ""
      result += str(data[1][0]) + "\t"
      result += str(data[1][1]) + "\t"
      result += str(data[0]) + "\t"
      result += str(data[1][2]) + "\t"
      result += str(data[1][3]) + "\t"
      result += str(data[1][4]) + "\t"
      result += str(data[1][5]) + "\t"
      result += str(data[1][6]) + "\t"
      return result
```

(b) Visualize these sessions in CartoDB (more info below)

We visualized the sessions in CartoDB with a subset of the data. Below is a screenshot. The check-ins from the same sessions share the same color.