ASS1 Brief Report

Yulin Huang 201676465

All codes run on Original Ubuntu Linux (dual operating system) using Python 3.6.13 and Pyspark 2.4.6

Task 1:

Resource Used:

1. ***PySpark Read CSV file into DataFrame***

   From:

   https://sparkbyexamples.com/pyspark/pyspark-read-csv-file-into-dataframe/

2. ***PySpark SQL – Working with Unix Time | Timestamp***

   From:

   https://sparkbyexamples.com/pyspark/pyspark-sql-working-with-unix-time-timestamp/

3. ***PySpark Concatenate Columns***

   From:

   https://sparkbyexamples.com/pyspark/pyspark-concatenate-columns/

4. ***PySpark date_format() – Convert Date to String format***

   From:

   https://sparkbyexamples.com/pyspark/pyspark-date_format-convert-date-to-string-format/

Method used:

Read data section:

Notice:

I added this extra section:

```
df = df.withColumn("Date", F.date_format("Date",
"yyyy-MM-dd"))
```

This is to keep the date format correct (no "time" is included in the "date" to avoid miscalculations). This is because using the auto-inferred mode, the format of the "date" will be "yyyy-mm-dd hh-mm-ss", which contains Time part. And this will lead to confusing results in the task later on.

Task 1 section:

First, concatenating the date and time strings by executing the SQL command using the expr function. The unix_timestamp function is used to convert a "date and time" string in the format "yyyy-MM-dd HH:mm:ss" to a Unix timestamp representing the number of seconds since 1 January 1970, which is preparing for time zone conversions. The timestamp for the China time zone is then adjusted by adding 28,800 seconds (equivalent to 8 hours) and then converted to the standard timestamp format using cast("timestamp"). The timestamp is further converted to a string format using the F.date_format function, which generates the "yyyy-MM-dd" and "HH:mm:ss" representations that replace the previous date and time columns in the DataFrame. Finally, new China timestamp is updated by adding 8/24 day to the timestamp column.

Results:

```
Task 1:
+------+---------+----------+-------+----------------+------------------+----------+--------+
|UserID| Latitude| Longitude|AllZero|        Altitude|         Timestamp|      Date|    Time|
+------+---------+----------+-------+----------------+------------------+----------+--------+
|   130|39.975088| 116.33269|      0|492.126135170604| 40001.13300925923|2009-07-07|03:11:32|
|   130| 39.97504|116.332806|      0|491.743412073491| 40001.13302083334|2009-07-07|03:11:33|
|   130|39.975009|116.332997|      0|491.676630577428| 40001.13303240744|2009-07-07|03:11:34|
|   130|39.975048|116.332932|      0|491.403044619423|40001.133043981434|2009-07-07|03:11:35|
|   130|39.974977| 116.33305|      0|491.043900918635| 40001.13306712964|2009-07-07|03:11:37|
|   130|39.974967|116.333116|      0|489.435272309711| 40001.13318287033|2009-07-07|03:11:47|
|   130|39.974931|116.333188|      0|487.797303149606| 40001.13329861114|2009-07-07|03:11:57|
|   130|39.974927|116.333249|      0|489.250744750656| 40001.13336805553|2009-07-07|03:12:03|
|   130|39.974953| 116.33331|      0|487.910994094488| 40001.13347222224|2009-07-07|03:12:12|
|   130|39.974988|116.333359|      0|486.979045275591| 40001.13353009264|2009-07-07|03:12:17|
|   130|39.975029|116.333311|      0|486.138612204724| 40001.13361111114|2009-07-07|03:12:24|
|   130|39.974993|116.333368|      0|485.249717847769| 40001.13365740744|2009-07-07|03:12:28|
|   130|39.975013|116.333424|      0|484.258648293963| 40001.13373842594|2009-07-07|03:12:35|
|   130|39.975008|  116.3335|      0|483.536105643045| 40001.13379629634|2009-07-07|03:12:40|
|   130|39.975018|116.333564|      0|482.802204724409|40001.133854166634|2009-07-07|03:12:45|
|   130|39.974994|116.333631|      0|482.616423884514| 40001.13386574073|2009-07-07|03:12:46|
|   130|   39.975|116.333703|      0|482.115708661417| 40001.13388888894|2009-07-07|03:12:48|
|   130|39.975025|116.333767|      0|481.127742782152| 40001.13396990744|2009-07-07|03:12:55|
|   130|39.974989|116.333882|      0|480.362650918635|40001.134027777734|2009-07-07|03:13:00|
|   130|39.974964|116.333959|      0|480.077047244094| 40001.13405092594|2009-07-07|03:13:02|
+------+---------+----------+-------+----------------+------------------+----------+--------+
only showing top 20 rows
```

Task 2:

Resource Used:

1. ***PySpark Groupby Agg (aggregate) – Explained***

   From:

   https://sparkbyexamples.com/pyspark/pyspark-groupby-agg-aggregate-explained/

Method used:

First, calculate the number of data points recorded per user per day. Then, use the groupBy function to group the DataFrame by "UserID" and "Date" and then count the records in each group. The results are placed in the "count_each_usr_a_day" DataFrame. The "at_least_five_records" DataFrame is created by applying a filter to the "count_each_usr_a_day" column based on a condition that checks if the "count" column is greater than or equal to 5 to identify if the user has submitted records for 5 or more data points in a day. Then grouped by "UserID" and calculat a unique date count for each user, using the agg function to rename the column to "DaysWithAtLeastFiveRecords", resulting in the "usr_ID_and_count" data frame. To identify data frames containing at least the five highest days records, sort the "usr_ID_and_count" data frame by "DaysWithAtLeastFiveRecords" in descending order and "UserID" in ascending order to generate the "top_users" data frame. Finally, use the limit function to limit the result to the first 5 users and return the "top_five" data frame as the final result.

Results:

```
Task 2:
+------+--------------------------+
|UserID|DaysWithAtLeastFiveRecords|
+------+--------------------------+
|   153|                      1251|
|   144|                       340|
|   140|                       237|
|   142|                       108|
|   134|                        57|
+------+--------------------------+
```

Task 3:

Resource Used:

1. ***weekofyear() returning seemingly incorrect results for January 1***
   From:
   https://stackoverflow.com/questions/49904570/weekofyear-returning-seemingly-incorrect-results-for-january-1

2. ***PySpark Count Distinct from DataFrame***
   From:
   https://sparkbyexamples.com/pyspark/pyspark-count-distinct-from-

[dataframe/]()

Method used:

First, adjust the timestamp values by subtracting 2 from the original timestamp. This calculation could set 1th January 1900 (Monday) as the reference start point to avoid mess caused by handling year boundaries. Next, divide this adjusted timestamp by 7 to calculate the number of weeks since 30th December 1899 (Saturday). In this case, counting weeks from 1st January 1900 (Monday) is equivalent to counting weeks from 30th December 1899 (Saturday).

Apply a filter to keep records where a user submitted more than 100 data points in a week, and create the "more_than_100_records" data frame.

Group the "more_than_100_records" data frame by "UserID" and count the unique "WhichWeek" values using the countDistinct function. Store this count in a new column named "WeeksMoreThan100Record", which is the number of weeks each user submitted 100 data points.

Sort the results in ascending order based on "UserID" to complete the process.

Notice:

Using the weekofyear() function will fail to handle the case of inter-annual weeks, resulting in incorrect results

Results:

```
Task 3:
+------+---------------------+
|UserID|WeeksMoreThan100Record|
+------+---------------------+
|   130|                    9|
|   131|                    9|
|   132|                    2|
|   133|                    4|
|   134|                   15|
|   135|                    6|
|   136|                    4|
|   137|                    1|
|   138|                    6|
|   139|                    4|
|   140|                   52|
|   141|                    8|
|   142|                   31|
|   143|                    1|
|   144|                   63|
|   145|                    5|
|   146|                    2|
|   147|                   12|
|   148|                    3|
|   149|                    1|
+------+---------------------+
only showing top 20 rows
```

Task 4:

Resource Used:

1. ***pyspark.sql.DataFrame.join***

   From:

   https://spark.apache.org/docs/3.1.2/api/python/reference/api/pyspark.sql.DataFrame.join.html

2. ***PySpark Join Types | Join Two DataFrames***

   From:

   https://sparkbyexamples.com/pyspark/pyspark-join-explained-with-examples/

Method used:

```python
def task_4(dataframe):
    # Get the minimum latitude for each user on each day
    min_latitudes_per_user_per_day = dataframe.groupBy("UserID", "Date").agg(F.min("Latitude").alias("MinLatitude"))
    # min_latitudes_per_user_per_day.show()

    # Get the overall minimum latitudes of each user (which is the Most Southern Latitude),by groupby userID again and get the minimum latitude
    min_latitudes_per_user = min_latitudes_per_user_per_day.groupBy("UserID").agg(
        F.min("MinLatitude").alias("MostSouthernLatitude"))
    # min_latitudes_per_user.show()

    # Rename the column in "min_latitudes_per_user_per_day",ready for joining
    # In this situation "MostSouthernLatitude" is not the real overall MostSouthernLatitude, it's just for joining and matching
    new_min_latitudes_per_day = min_latitudes_per_user_per_day.withColumnRenamed("MinLatitude", "MostSouthernLatitude")
    # new_min_latitudes_per_day.show()

    # Join the dataframes with "UserID" (primary),
    # and use the "overall" MostSouthernLatitude in "min_latitudes_per_user" to join the "Fake" MostSouthernLatitude Colunm in new_min_latitudes_per_day
    # Let the "overall" MostSouthernLatitude to match the "MinLatitude"(Which is the "Fake" MostSouthernLatitude Colunm now) in min_latitudes_per_user_per_day
    # Then will get a form with all information needed
    southernmost_dates = new_min_latitudes_per_day.join(min_latitudes_per_user, ["UserID", "MostSouthernLatitude"])
    # southernmost_dates.show()

    # Get the records with the earliest date
    southernmost_dates = southernmost_dates.groupBy("UserID", "MostSouthernLatitude").agg(
        F.min("Date").alias("FirstDateOfMostSouthern"))

    # Get top 5 users
    top_5_most_southern_users = southernmost_dates.orderBy("MostSouthernLatitude", "UserID").limit(5)

    return top_5_most_southern_users
```

First, create the "min_latitudes_per_user_per_day" DataFrame with calculating the minimum latitude each user records each day. Next, find each user's overall minimum latitude (the southernmost latitude) by grouping the "min_latitudes_per_user_per_day" DataFrame by "UserID" and calculating the minimum latitude for each group. Store this information in the "min_latitudes_per_user" DataFrame with the column name "MostSouthernLatitude".

Rename the "MinLatitude" column in "min_latitudes_per_user_per_day" to "MostSouthernLatitude" to prepare for joining operations.

Notice:

```
In this situation "MostSouthernLatitude" is not the
real overall MostSouthernLatitude, it's just for
joining and matching.
```

```
Join the dataframes with "UserID" (primary),
and use the "overall" MostSouthernLatitude in
"min_latitudes_per_user" to join the "Fake"
MostSouthernLatitude Colunm in
new_min_latitudes_per_day.

Let the "overall" MostSouthernLatitude to match the
"MinLatitude"(Which is the "Fake"
MostSouthernLatitude Colunm now) in
min_latitudes_per_user_per_day
```

```python
# Rename the column in "min_latitudes_per_user_per_day",ready for joining
# In this situation "MostSouthernLatitude" is not the real overall MostSouthernLatitude, it's just for joining and matching
new_min_latitudes_per_day = min_latitudes_per_user_per_day.withColumnRenamed("MinLatitude", "MostSouthernLatitude")
# new_min_latitudes_per_day.show()

# Join the dataframes with "UserID" (primary),
# and use the "overall" MostSouthernLatitude in "min_latitudes_per_user" to join the "Fake" MostSouthernLatitude Colunm in new_min_latitudes_per_day
# Let the "overall" MostSouthernLatitude to match the "MinLatitude"(Which is the "Fake" MostSouthernLatitude Colunm now) in min_latitudes_per_user_per_day
# Then will get a form with all information needed
southernmost_dates = new_min_latitudes_per_day.join(min_latitudes_per_user, ["UserID", "MostSouthernLatitude"])
# southernmost_dates.show()
```

Then select each user's earliest date record and its corresponding southernmost latitude. Grouping the "South_most_dates" data frames by "User_ID" and "South_most_latitude" and then calculating the minimum date. Then will get the column "FirstDateOfMostSouthern".

Finally, sort the results in ascending order based on "MostSouthernLatitude" and "UserID" and limit the output to the first 5 users with the most southern latitude records.

<u>Results:</u>

```
Task 4:
+------+--------------------+----------------------+
|UserID|MostSouthernLatitude|FirstDateOfMostSouthern|
+------+--------------------+----------------------+
|   144|            1.044024|            2009-03-25|
|   142|           1.2724579|            2008-07-26|
|   160|           13.359522|            2010-12-27|
|   153|        22.230673974|            2012-05-08|
|   134|    22.3335333333333|            2007-07-18|
+------+--------------------+----------------------+
```

Task 5:

Resource Used:

1. ***Feet to Meters Converter***

   From :

   https://www.rapidtables.com/convert/length/feet-to-meter.html

Method used:

First, create a DataFrame called span to calculate the height difference for each user for each day. Calculate this difference by subtracting the minimum altitude from the maximum altitude for each user each day. Convert the span value from feet to metres. Next, aggregate these spans into a DataFrame named max_span to find the maximum altitude span for each user on a single day. This is done by grouping the data by user ID and applying the F.max function to the altitude span. Sort the users by their maximum altitude spans in a DataFrame named top_five_span_users. Then, select the top 5 users based on the greatest altitude spans they have in a single day.

Results:

```
Task 5:
+------+-----------------+
|UserID|          MAXSpan|
+------+-----------------+
|   144|26217.006240000002|
|   140|       16710.99528|
|   153| 7723.200042999985|
|   147|       2142.249395|
|   142|         1979.9808|
+------+-----------------+
```

Task 6:

Resource Used:

1. ***Getting distance between two points based on latitude/longitude***

   From :

   https://stackoverflow.com/questions/19412462/getting-distance-between-two-points-based-on-latitude-longitude

2. ***Spark SQL Row_number() PartitionBy Sort Desc***
   From:
   https://stackoverflow.com/questions/35247168/spark-sql-row-number-partitionby-sort-desc

3. ***PySpark UDF (User Defined Function)***
   From:
   https://sparkbyexamples.com/pyspark/pyspark-udf-user-defined-function/

4. ***User-defined scalar functions – Python***
   From:
   https://docs.databricks.com/en/udf/python.html

5. ***Applying a Window function to calculate differences in pySpark***
   From:
   https://stackoverflow.com/questions/36725353/applying-a-window-function-to-calculate-differences-in-pyspark

6. ***SQL Window Functions vs. GROUP BY: What's the Difference?***
   From:
   https://learnsql.com/blog/sql-window-functions-vs-group-by/

7. ***PySpark Lag***
   From:
   https://www.educba.com/pyspark-lag/

Method used:

Function distance(lon1, lat1, lon2, lat2):

Notice:
Using different Earth radii can bias the results, e.g. a search in Google gives an Earth radius of 6373km, but the provided stack overflow gives an Earth radius of 6371km.

Applie the Haversine formula to calculate the great-circle distance between two points on the Earth's surface, based on their longitude and latitude in degrees. First, it converts these degrees to radians, which is essential for the trigonometric calculations in the formula. The formula then computes the central angle between the points, using an approximation of the Earth's radius as 6371.0 kilometers (a value from Google, though some sources like Stack Overflow suggest 6373 km). Finally, it calculates the distance in kilometers between the two points by using the differences in latitude and longitude and applying trigonometric functions to these differences.

Function task_6(dataframe):

Notice:
In the process of defining UDFs, specifying different data types can affect the results to different degrees, here I use Doubletype to get a higher accuracy in calculations

```
# PySpark UDF's are similar to UDF on traditional databases.
# In PySpark, you create a function in a Python syntax and wrap it with PySpark SQL udf() or register it as udf and use it on DataFrame and SQL respectively.
# Explanation From https://sparkbyexamples.com/pyspark/pyspark-udf-user-defined-function/
# Use DoubleType to get the most accurate results
distance_udf = F.udf(distance, DoubleType())
```

First, create a user-defined function (distance_udf) with the distance function that defined before. It could be used as a Dataframe function. Set a window specification, sorted by timestamp, for each UserID and Date to calculate the lag in latitude and longitude, get the previous records for each point. Aggregate this data to determine each user's total daily travel distance. Sort these totals to get the day with the longest travel distance for each user, by applying a window function that sorts by largest to smallest travel distance and earliest to latest date. Then, calculate the total distance traveled by all users on all dates. The final output includes the specific dates when each user traveled the most and the overall distance traveled by all users.

Notice:
If not handle the data that without previous record, there will be a NoneType Error. Then I choose to remove those data.

```
# To avoid NoneType error, need to remove the Data that has no previous record
both_diff = both_diff.filter(dataframe["LastLatitude"].isNotNull() & dataframe["LastLongitude"].isNotNull())
```

Results:

```
Task 6:
The furthest day each user has travelled:
+------+----------+
|UserID|      Date|
+------+----------+
|   130|2009-09-12|
|   131|2009-04-21|
|   132|2010-05-01|
|   133|2011-04-21|
|   134|2007-07-07|
|   135|2009-01-25|
|   136|2008-05-30|
|   137|2011-01-28|
|   138|2007-06-27|
|   139|2007-10-04|
|   140|2009-01-15|
|   141|2011-10-23|
|   142|2008-05-07|
|   143|2009-09-12|
|   144|2009-03-26|
|   145|2008-04-30|
|   146|2007-08-01|
|   147|2011-03-06|
|   148|2011-05-15|
|   149|2009-09-12|
+------+----------+
only showing top 20 rows

Total distance travelled by all users on all days:  195588.5596035513 km
```

Task 7:

Resource Used:

1. ***PySpark When Otherwise | SQL Case When Usage***
   From:
   https://sparkbyexamples.com/pyspark/pyspark-when-otherwise/

2. ***Pyspark calculated field based off time difference***
   From:
   https://stackoverflow.com/questions/64254937/pyspark-calculated-field-based-off-time-difference

3. ***Pyspark - calculate average velocity in group based on time and location***
   From:
   https://stackoverflow.com/questions/62105465/pyspark-calculate-average-velocity-in-group-based-on-time-and-location

Method used:

In task 7, the function is the same as task 6 before calculating time diffenence and speed. Therefore repetitions are not described.

Calculate the time difference between two records by using the timestamp lag for each user on each day and converts this time difference from days to hours.

Then rank the records by descending speed for each user and filters out the record with the highest speed, get the earliest day each user reached their maximum speed. Returns the record with the highest speed and the earliest day each user reached the highest speed.

Notice:
There may be a problem with handling duplicate records with same timestamp.
I handled it by checking whether TimeDifference is equal to 0.

```
# Calculate the speed, as km/hours format
# Not sure if there are duplicate records with same timestamp, which may cause denominator be 0
# dataframe = dataframe.withColumn("Speed", F.col("Distance") / F.col("TimeDifference"))
# Check whether TimeDifference is equal to 0
# From PySpark When Otherwise | SQL Case When Usage
# https://sparkbyexamples.com/pyspark/pyspark-when-otherwise/
with_speed = time_diff.withColumn("Speed", F.when(F.col("TimeDifference") != 0,
                                               F.col("Distance") / F.col("TimeDifference")).otherwise(0))
```

Results:

```
Task 7:
+------+----------+-----------------+
|UserID|      Date|            Speed|
+------+----------+-----------------+
|   130|2009-09-05|487.04625075548785|
|   131|2009-07-16|234.48440320665648|
|   132|2010-05-01| 211.9339450641081|
|   133|2011-01-31|2090.0441080181563|
|   134|2007-07-07|1255.3720587591554|
|   135|2009-01-25|411.65681008229717|
|   136|2008-05-12| 324.5759022538471|
|   137|2011-01-28| 287.4237056545262|
|   138|2007-06-27| 7639.732205870548|
|   139|2007-09-04|1469.4030718537265|
|   140|2008-09-03| 5692.303567343548|
|   141|2011-08-31|1633.8552082653243|
|   142|2007-06-13|2020.3134718666638|
|   143|2009-09-13| 122.0335299929746|
|   144|2009-03-25|1171151.5934952013|
|   145|2008-05-24|203.25975185220844|
|   146|2007-08-01| 240.2506213643785|
|   147|2011-03-01| 420.6121919267796|
|   148|2011-05-15| 512.8981704557093|
|   149|2009-09-13| 479.9979417313806|
+------+----------+-----------------+
only showing top 20 rows
```