Large-scale Student Walking Data Analytics with Parallel Technology

Sirapat Na Ranong, 7 Aug 2017

Abstract – We have a large-scale of student walking data. However, it's hard to handle this huge amount of data and takes too long to complete the tasks. Thus, we study about big data and parallel technologies to handle this data.
We compare and discuss the differences between technologies. We describe the method of calculation, visualization, traffic mode estimation, and activity area recognition under the Spark framework. (One of a big data processing framework)

I. INTRODUCTION

We have a big data of student walking trajectory. The problem is that the data is too big so we use parallel technology to solve the problem. The traffic mode of the data is already estimated. However, we found that it's not accurate enough. Our purpose is to improve the traffic mode of this data using parallel technology. Thus, the data will be more correct and small sized for the next researchers which will be easier for further research.

II. BACKGROUND

The personal activity can be collected using the mobile sensors, such as mobile phone or other GPS devices, which can record the detailed trajectory of the people. Using this type of data, we can look insight into the urban mobility, and make city more livable and sustainable.

In Singapore, one large-scale experiment, named the National Science Experiment (NSE), is conducted to capture the young Singaporeans mobility using the mobile sensors. It targets to recruit more than 250,000 students carrying the SENSg device over four weeks. The SENSg sensor device measures and stores the student data of his/her motion, ambient temperature, humidity, atmospheric pressure, light intensity and sound pressure levels based on the sensor's location. And the smart sensor device uses Wi-Fi signals when it closes to Wi-Fi hotspots to localize itself. The sensor data is stored in sensor temporally and will be periodically uploaded to a cloud-based database anonymously if it is in range of a known access point. From this data, the travel demand of the students can be extracted.

However, it is quite challenging to forecast the travel demand from this type data because of the large size of this data. For example, the size of the student trajectory data collected by the experiment mentioned above is more than 30GB, which is difficult to process using the traditional method. So, we need to develop an approach to estimate the students travel demand based on the big data from the mobile sensors.

III. PARALLEL TECHNOLOGIES

Nowadays, The data is getting larger and larger. Serial computing takes too long to complete the job which is unsatisfied. To handle this big data in a limited time, we need parallel computing. Parallel computing is the simultaneous use of multiple resources to complete the job in an appropriate time. The resources include CPU Cores, RAM, and even GPU Cores.

However, multiple resources in a single machine may still not enough to process a huge amount of data. Introducing to Cluster computing. A cluster is a group of multiple standalone machines that are connected in a network. The purpose of a cluster is to leverage more power of computation since a single machine isn't enough. There are many parallel technologies that support cluster. Here is the example of those technologies supporting python language, **MPI, OPENMP, CUDA, Dask.distributed, Hadoop, Spark,** etc**.**

MPI [1] stands for Message Passing Interface. It is a communication protocol for parallel programming. It allows a single computer and also multiple computers in a cluster to use multiple cores to compute or perform a task in parallel. PyMpi and mpi4py are the well-known implementation of MPI in Python.

OPENMP [2] stands for Open Multi-Processing. It is an API for writing the multithreaded program. Threads communicate by sharing variables on the shared memory. Cython is a notable implementation of OPENMP in Python.

CUDA [3] is a parallel computation API model created by NVIDIA. It allows us to write the parallel program by using CUDA cores of the GPU instead of CPU cores. PyCUDA lets you access CUDA API by using Python.

Dask.distributed [4] is a library for distributed computing in Python. It helps us to write the code for communication between nodes in a cluster. It consists of scheduler and multiple workers. This library is very new. (2016-2017) [4]

Hadoop [5] is an open-source framework by Apache that is designed and created for processing and storing a large amount of data in a distributed file system called HDFS (Hadoop Distributed File System). MapReduce is a technique used by Hadoop to process a large amount of data in parallel by map and reduce function. It is java based but also supports Python.

Spark [6] is the cluster distributed computation technology developed by Apache. It is based on MapReduce but increases the performance by using memory instead of disk to store the data in each stage.

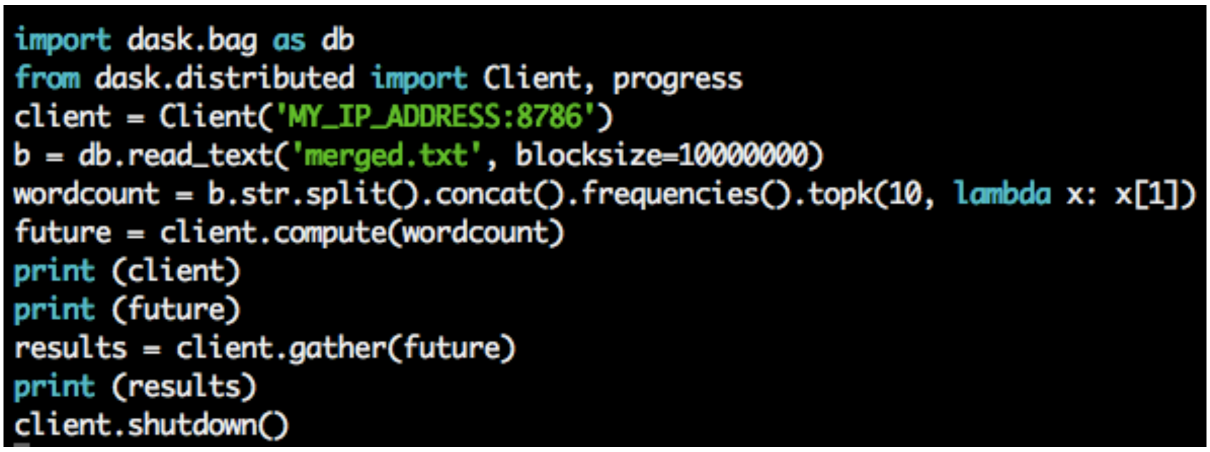
Table 1 Comparison of the distribution frameworks

Technologies	Main Resources	Model	Pros	Cons
MPI	CPU cores	Distributed Memory	Good for task parallelism	Hard to code/setup Bad for big data
OPENMP	CPU threads	Shared Memory	Easy to code	Not scalable Not suite for cluster
CUDA	GPU cores	Distributed Memory	High performance	Very hard to code Need high-end graphic card
Dask.distributed	CPU cores/threads	Distributed Memory	Lightweight Easy to setup	Less community support Bad for sorting, merges, subselecting column
MapReduce	CPU cores/threads	Distributed Memory	Handle big data Fault-tolerance Good for data parallelism	Hard to setup Intensive disk usage
Spark	CPU cores/threads Memory	Distributed Memory	Handle big data Fault-tolerance Good for data parallelism Faster speed	Hard to setup Consume large amount of memory

Testing - We ignore MPI, OPENMP, and CUDA since there are many other technologies that have more performance. My computer also has limited resources to run many VMs. The testing environment is VMs of Ubuntu server with only 1 core and 1GB RAM for each machine. The input file is 76MB.

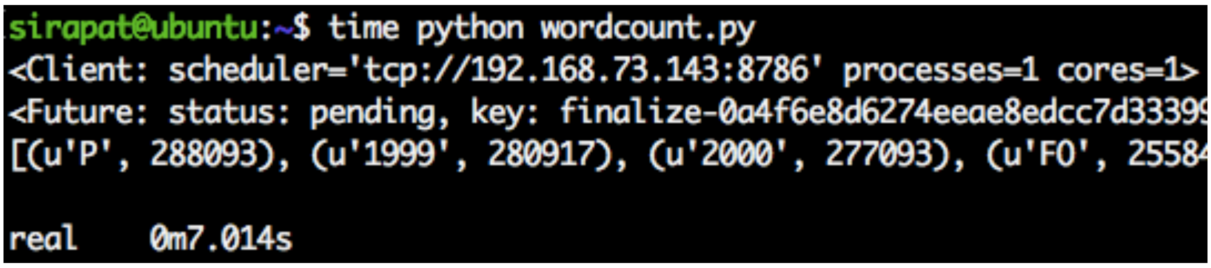
Dask.distributed - running word count with the input file

Testing code – wordcount.py modified from [9]



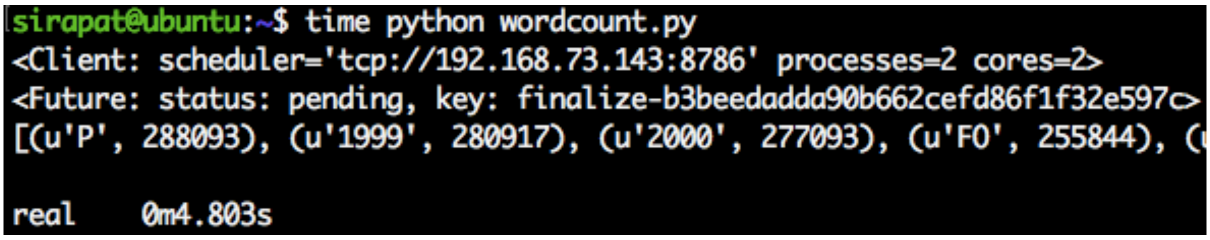
```
import dask.bag as db
from dask.distributed import Client, progress
client = Client('MY_IP_ADDRESS:8786')
b = db.read_text('merged.txt', blocksize=10000000)
wordcount = b.str.split().concat().frequencies().topk(10, lambda x: x[1])
future = client.compute(wordcount)
print (client)
print (future)
results = client.gather(future)
print (results)
client.shutdown()
```

1 machine took 7 seconds.



```
sirapat@ubuntu:~$ time python wordcount.py
<Client: scheduler='tcp://192.168.73.143:8786' processes=1 cores=1>
<Future: status: pending, key: finalize-0a4f6e8d6274eeae8edcc7d3339
[(u'P', 288093), (u'1999', 280917), (u'2000', 277093), (u'FO', 2558

real	0m7.014s
```

2 machines took 4.8 seconds.

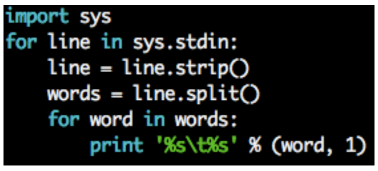


```
sirapat@ubuntu:~$ time python wordcount.py
<Client: scheduler='tcp://192.168.73.143:8786' processes=2 cores=2>
<Future: status: pending, key: finalize-b3beedadda90b662cefd86f1f32e597c>
[(u'P', 288093), (u'1999', 280917), (u'2000', 277093), (u'FO', 255844), (

real	0m4.803s
```

Hadoop - running word count with the same input file

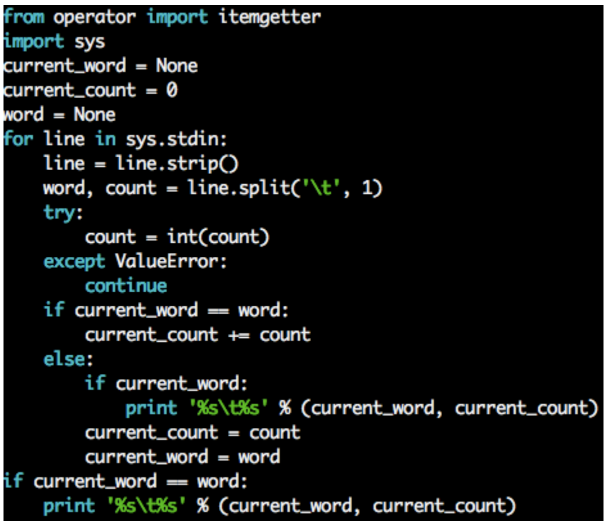
Testing code – mapper.py, reducer.py [10]

mapper.py

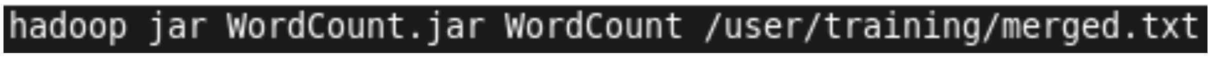


```
import sys
for line in sys.stdin:
    line = line.strip()
    words = line.split()
    for word in words:
        print '%s\t%s' % (word, 1)
```

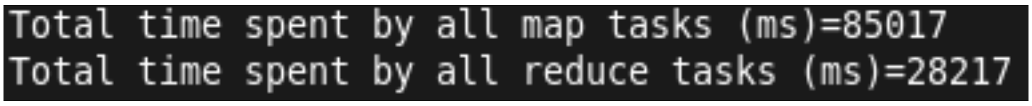
reducer.py



```
from operator import itemgetter
import sys
current_word = None
current_count = 0
word = None
for line in sys.stdin:
    line = line.strip()
    word, count = line.split('\t', 1)
    try:
        count = int(count)
    except ValueError:
        continue
    if current_word == word:
        current_count += count
    else:
        if current_word:
            print '%s\t%s' % (current_word, current_count)
        current_count = count
        current_word = word
if current_word == word:
    print '%s\t%s' % (current_word, current_count)
```



```
hadoop jar WordCount.jar WordCount /user/training/merged.txt
```

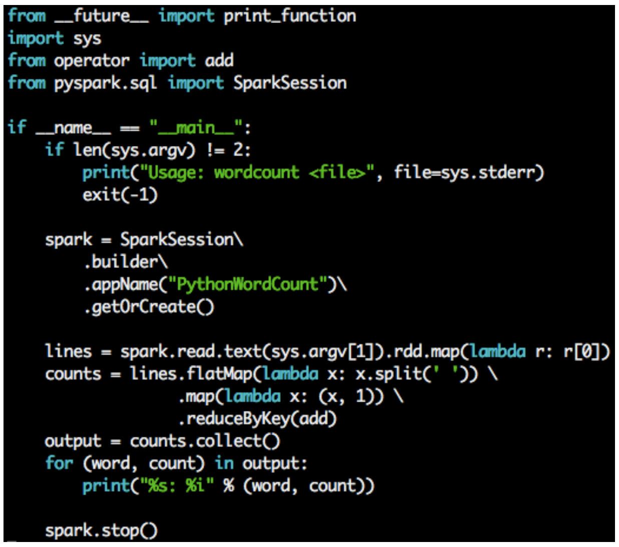


```
Total time spent by all map tasks (ms)=85017
Total time spent by all reduce tasks (ms)=28217
```

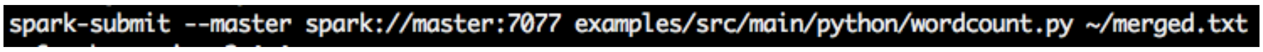
Total about 113 seconds including overhead and real task.

Spark - running word count with the same input file

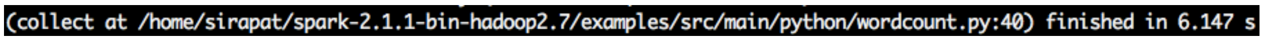
Testing code – wordcount.py from Spark-2.1.1 example



```
from __future__ import print_function
import sys
from operator import add
from pyspark.sql import SparkSession

if __name__ == "__main__":
    if len(sys.argv) != 2:
        print("Usage: wordcount <file>", file=sys.stderr)
        exit(-1)

    spark = SparkSession\
        .builder\
        .appName("PythonWordCount")\
        .getOrCreate()

    lines = spark.read.text(sys.argv[1]).rdd.map(lambda r: r[0])
    counts = lines.flatMap(lambda x: x.split(' ')) \
                  .map(lambda x: (x, 1)) \
                  .reduceByKey(add)
    output = counts.collect()
    for (word, count) in output:
        print("%s: %i" % (word, count))

    spark.stop()
```



```
spark-submit --master spark://master:7077 examples/src/main/python/wordcount.py ~/merged.txt
```



```
(collect at /home/sirapat/spark-2.1.1-bin-hadoop2.7/examples/src/main/python/wordcount.py:40) finished in 6.147 s
```

It took about 6 seconds to do the task. However, there is overhead about 60-80 seconds to prepare, schedule, handle faults in the system. Due to limited resources of my computer. I can't use pretty big data to test it. Thus, Dask.distributed will be the best for this small data. However, for 100GB+ data, the overhead of 60-100 seconds is not much for a big job. We think it is worth to use Hadoop or Spark to process because they are fault-tolerance which is very important for big data. They provide more features, supported libraries and dedicated data structure (Ex.RDD). Moreover, they have a big community for developers. Spark is a better choice than MapReduce if you have enough memory. Dask.distributed would be preferred if you don't have access to cluster because it is lightweight.

IV. APACHE SPARK INSTALLATION ON A CLUSTER AND ENVIRONMENT CONFIGURATION

***Prerequisites** (for every single machine on the cluster)*

1. *Java environment*

   Verify that Java is installed.

   ```
   $ java –version
     # java version "1.8.0_91"
     # Java(TM) SE Runtime Environment (build 1.8.0_91-b14)
     # Java HotSpot(TM) 64-Bit Server VM (build 25.91-b14, mixed mode)
   ```

   Java is already installed.

2. *Scala environment*

   Verify that Scala is installed.

   ```
   $ scala –version
     # scala: command not found
   ```

   Scala isn't installed yet. Thus, download Scala and then install it.

   ```
   $ wget http://downloads.lightbend.com/scala/2.12.2/scala-2.12.2.rpm
   ```

   ```
   $ yum install scala-2.12.2.rpm
   ```

***Install Spark on master** (we will copy it to slaves later)*

1. *Download latest version of Spark*

   ```
   $ wget https://d3kbcqa49mib13.cloudfront.net/spark-2.1.1-bin-hadoop2.7.tgz
   ```

2. *Untar it*

   ```
   $ tar xzf spark-2.1.1-bin-hadoop2.7.tgz
   ```

3. *Rename to spark and move to /usr/*

   ```
   $ mv spark-2.1.1-bin-hadoop2.7 /usr/spark
   ```

***Configuration** (for every single machine on the cluster or you can do it once in a machine and then copy to the others by scp)*

1. *Edit .bashrc file*

   ```
   $ nano ~/.bashrc
   ```

   Add the following lines.

   ```
   export JAVA_HOME=/opt/jdk1.8.0_91/
   export SPARK_HOME=/usr/spark
   export PATH=$PATH:$SPARK_HOME/bin
   export PATH=$PATH:$HOME/bin:$JAVA_HOME/bin
   ```

   Then source the .bashrc file.

   ```
   $ . ~/.bashrc   # alternative: $ source ~./bashrc
   ```

2. *Copy "slaves.template" file in /usr/spark/conf/ to "slaves" file*

   ```
   $ cp slaves.template slaves
   ```

   Edit "slaves" file in /usr/spark/conf/

   ```
   $ nano /usr/spark/conf/slaves
   ```

   Add the following lines.

   ```
   slave1  # this is the IP address of VM-02 that is configured in /etc/hosts
   slave2  # this is the IP address of VM-03 that is configured in /etc/hosts
   slave3  # this is the IP address of VM-04 that is configured in /etc/hosts
   ```

3. Copy "spark-env.sh.template" file in /usr/spark/conf/ to "spark-env.sh" file

   ```
   $ cp spark-env.sh.template spark-env.sh
   ```

   Edit "spark-env.sh" file in /usr/spark/conf/

   ```
   $ nano /usr/spark/conf/spark-env.sh
   ```

   Add the following lines.

   ```
   export JAVA_HOME=/opt/jdk1.8.0_91/
   export HADOOP_CONF_DIR=$HADOOP_HOME/etc/hadoop
   export YARN_CONF_DIR=$HADOOP_HOME/etc/hadoop
   export SPARK_MASTER_IP=master
   export SPARK_LOCAL_IP=master
   export SPARK_MASTER_HOST=master
   ```

   Note: SPARK_LOCAL_IP is different for each machine configuration.

4. Make directory in hdfs and put all spark jars(in spark directory) into hdfs.

   ```
   $ hadoop dfs –mkdir –p spark_jar
   $ hadoop dfs –put jars/*.jar /spark_jar/
   ```

5. Copy spark-defaults.conf.template to spark-defaults.conf

   ```
   $ cp spark-defaults.conf.template spark-defaults.conf
   ```

6. Add this line in spark-defaults.conf

   ```
   spark.yarn.jars = hdfs:///user/root/spark_jar/*.jar
   ```

7. Copy spark from master to slaves.

   ```
   $ scp –r /usr/spark slave1:/usr/
   $ scp –r /usr/spark slave2:/usr/
   $ scp –r /usr/spark slave3:/usr/
   ```

   Make sure that SPARK_LOCAL_IP in /usr/spark/conf/spark-env.sh is correct.
   In every single machine, edit spark-env.sh file in /usr/spark/conf

   ```
   $ nano /usr/spark/conf/spark-env.sh
   ```

   Edit SPARK_LOCAL_IP to match the particular machine.

   For VM01 (master): export SPARK_LOCAL_IP=master
   For VM02 (slave1): export SPARK_LOCAL_IP=slave1
   For VM03 (slave2): export SPARK_LOCAL_IP=slave2
   For VM04 (slave3): export SPARK_LOCAL_IP=slave3

How to use

1. Start the Spark cluster.
   ```
   $ /usr/spark/sbin/start-all.sh
   ```
2. Check the services
   ```
   $ jps
   ```
3. Submitting the job in YARN mode
   ```
   $ spark-submit --master yarn --deploy-mode cluster code.py
   ```
   Submitting the job in Standalone mode (**This one is preferred**)
   ```
   $ spark-submit --master spark://master:7077 code.py
   ```
4. You can also enter the shell directly by `$ pyspark` for python or `$ spark-shell` for scala
5. Stop the Spark cluster
   ```
   $ /usr/spark/sbin/stop-all.sh
   ```

Monitoring

1. For job that is submitted in Standalone mode
   Spark Master UI is at https://master:8080
2. For job that is submitted in YARN mode
   Hadoop YARN at http://master:8088/cluster/app/application_ID
   Ex. http://172.20.98.198:8088/cluster/app/application_1497516107234_0003
3. See list of applications in YARN
   ```
   $ yarn application -list -appStates ALL
   ```
4. To be able to see logs we need to enable logging first by
   Edit yarn-site.xml in /usr/hadoop/hadoop-2.7.3/etc/hadoop/
   ```
   $ nano /usr/hadoop/hadoop-2.7.3/etc/hadoop/yarn-site.xml
   ```
   Add the following lines.
   ```
     <property>
         <name>yarn.log-aggregation-enable</name>
         <value>true</value>
     </property>
     <property>
        <name>yarn.nodemanager.remote-app-log-dir</name>
        <value>/app-logs</value>
     </property>
     <property>
         <name>yarn.nodemanager.remote-app-log-dir-suffix</name>
         <value>logs</value>
     </property>
   ```
5. See the log of an application
   ```
   $ yarn logs –applicationId APPLICATION_ID
   # Ex. $ yarn logs –applicationId application_1497516107234_0003
   ```
6. See the log manually
   ```
   $ hadoop dfs –cat /app-logs/root/logs/APPLICATION_ID/FILE
   # Ex. $ hadoop dfs –cat /app-logs/root/logs/1497516107234_0003/slave1_39218
   ```

V. SPARK PERFORMANCE TUNING

Resources we have:

Total machines: 3+1 (1 is Master)

Total cores: 45

Total memory: 81 GB

Each machine: 15 cores, 27 GB

According to the Spark guidelines[11], for this amount of resources, we should configure 3 workers for each machine to get the best performance

Each machine: 3 workers

Each worker: 5 cores, 9 GB

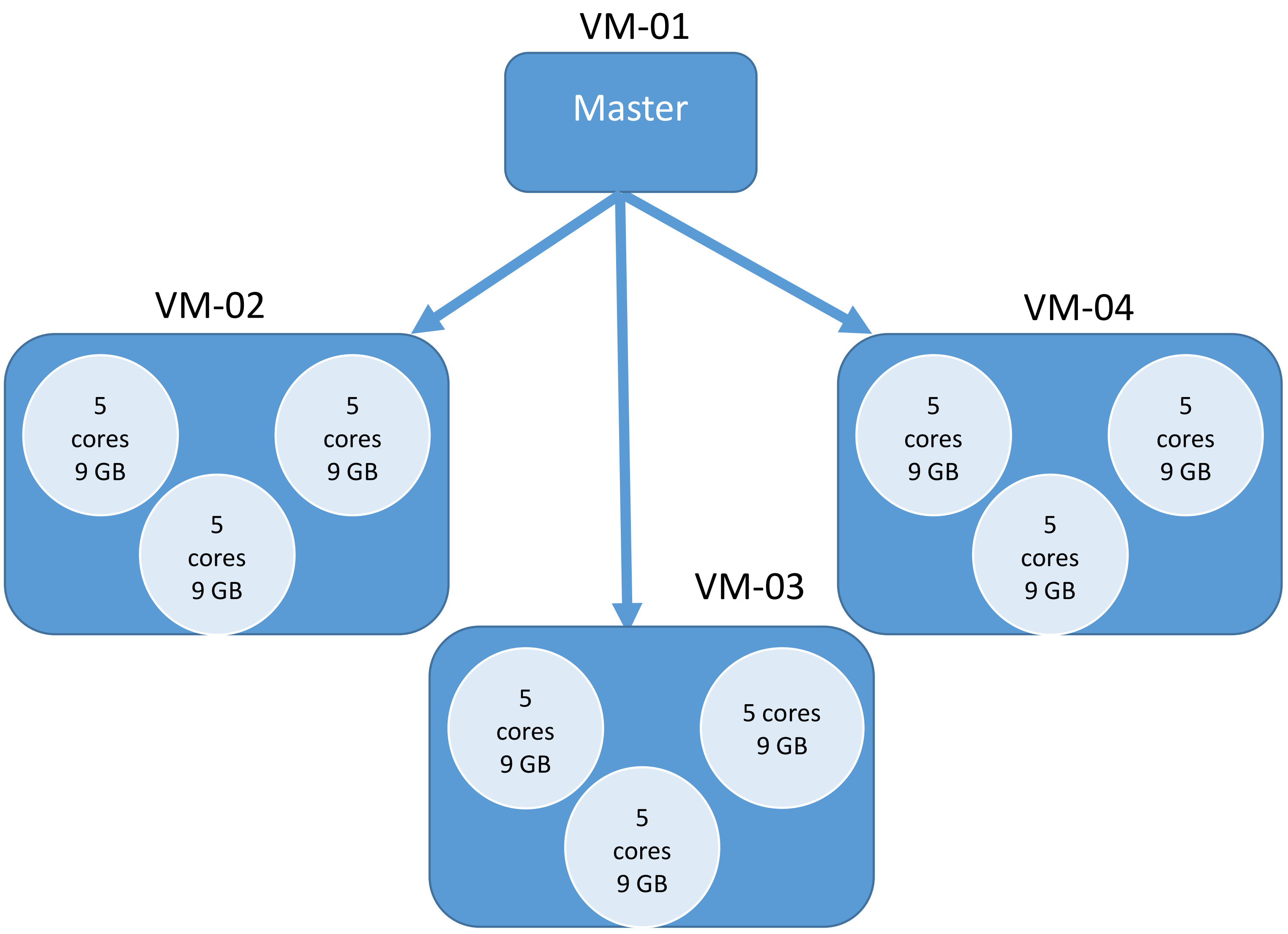


Figure 1. Spark cluster structure

VI. SPARK INTRODUCTION

Apache Spark is a fast and general-purpose cluster computing system. It provides high-level APIs in Java, Scala, Python and R, and an optimized engine that supports general execution graphs. It also supports a rich set of higher-level tools including Spark SQL for SQL and structured data processing, MLlib for machine learning, GraphX, and Spark Streaming. [12]

Pyspark is the Spark Python API that allows you to programming in Python under the framework of Spark. We use Pyspark because Python is a very powerful language. It can do tons of work within a few lines of code. The performance can be dropped comparing with Scala or Java. However, it's very faster to develop and understand a program with Python.

Data type in Spark

There are 3 main data types in Spark which are RDD, DataFrame, and DataSet. RDD is Resilient Distributed Dataset. It's a distributed collection which is the main data type in Spark. DataFrame and DataSet are newer data type. The differences between them is that DataSet errors will be in compile time but DataFrame errors will be in run time. However, DataFrame give higher performance especially on interactive analysis.

Which one should we use?

RDD if you are working with unstructured data. DataFrame or Dataset if you are working with structured data. We use DataFrame because we are working with structured data and concern about the best performance.

Where should the input data stored?

The data should be stored on HDFS so every machine on the cluster can access to it. If the data is in local file system, then you have to copy it to every local file system of every machine.

How to process big data with Pyspark?

To work with DataFrame, we will use pyspark.sql module. Here are some simple steps.

1. Create a spark session
```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("YOUR_APP_NAME").getOrCreate()
```
2. Read the input data to the DataFrame
```
df = spark.read.csv("PATH_TO_INPUT", header= True)
```
3. Common functions with DataFrame

   3.1 Cache/Clear the DataFrame in memory

   You will need this for interactive queries.
```
df.cache()
df.clearCache()
```

3.2 Remove the column

```
df = df.drop(“COLUMN_NAME”)
```

3.3 New or update the column if the column is existed

```
df = df.withColumn(“COLUMN_NAME”, COLUMN_OBJECT)
```

3.4 Filter the rows

```
df = df.filter(CONDITION)
```

4. IF, ELSE Condition

   from pyspark.sql.functions import when

```
df.when(CONDITION, RETURNED_VALUE).otherwise(RETURNED_VALUE)
```

5. Apply a function to the DataFrame

   You will need functions from pyspark.sql.funtions instead of native python library.

   Here is an example of applying function cosine to a column of every row of DataFrame.

```
from pyspark.sql.functions import cos
def cosine(number)
        return cos(number)
df = df.withColumn(“COLUMN_NAME”, cosine(df.number)
```

6. Write the output csv file

```
df.write.format('csv').save("OUTPUT_PATH",header=True)
```

VII. ANALYZE LARGE-SCALE STUDENT TRAJECTORY DATA WITH PYSPARK AND ORANGE

Here is the calculation for the first section of our analysis.

1. Calculate the time step
2. Calculate speed step in steps/second
3. Calculate the distance between two coordinate points by Haversine formula

$$d = 2r \arcsin\left(\sqrt{\sin^2\left(\frac{\phi_2 - \phi_1}{2}\right) + \cos(\phi_1)\cos(\phi_2)\sin^2\left(\frac{\lambda_2 - \lambda_1}{2}\right)}\right)$$

4. Calculate speed distance in meter/second

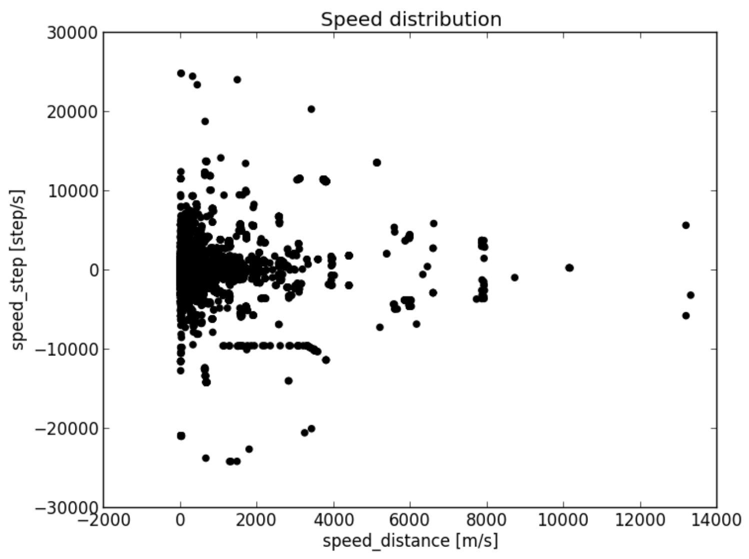


Figure 2. Relationship between speed_distance and speed_step

There are many unreasonable records from the data. Those are the records with negative or over speed. We found that the accuracy of the GPS is not enough. In a small time span, the trajectory is not smooth and the point may jump from position to position. With a GPS, even when you keep it static, the speed is also larger than 0 caused by data noise of GPS device. Thus, we set the threshold to remove those noises. After we set the time interval to 20-60 seconds, many unreasonable records have been removed. We then set the speed_step to 0-5 steps/second and speed_distance to 0-55 meters/second (0-198 km/hr) to improve the data.

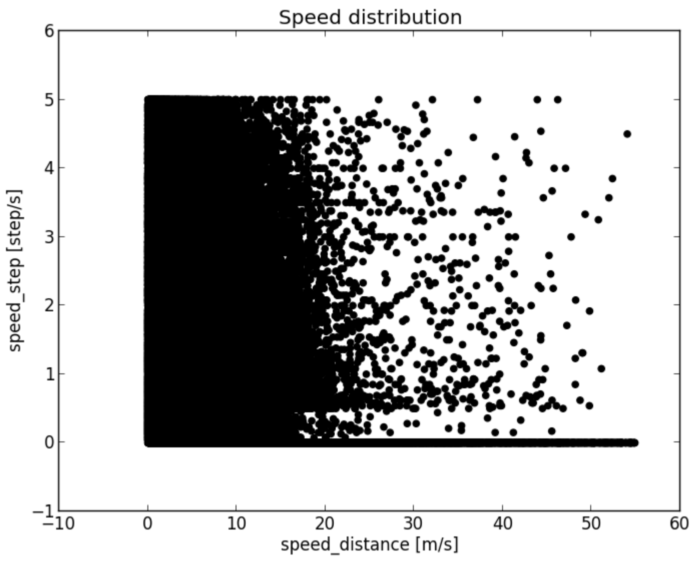
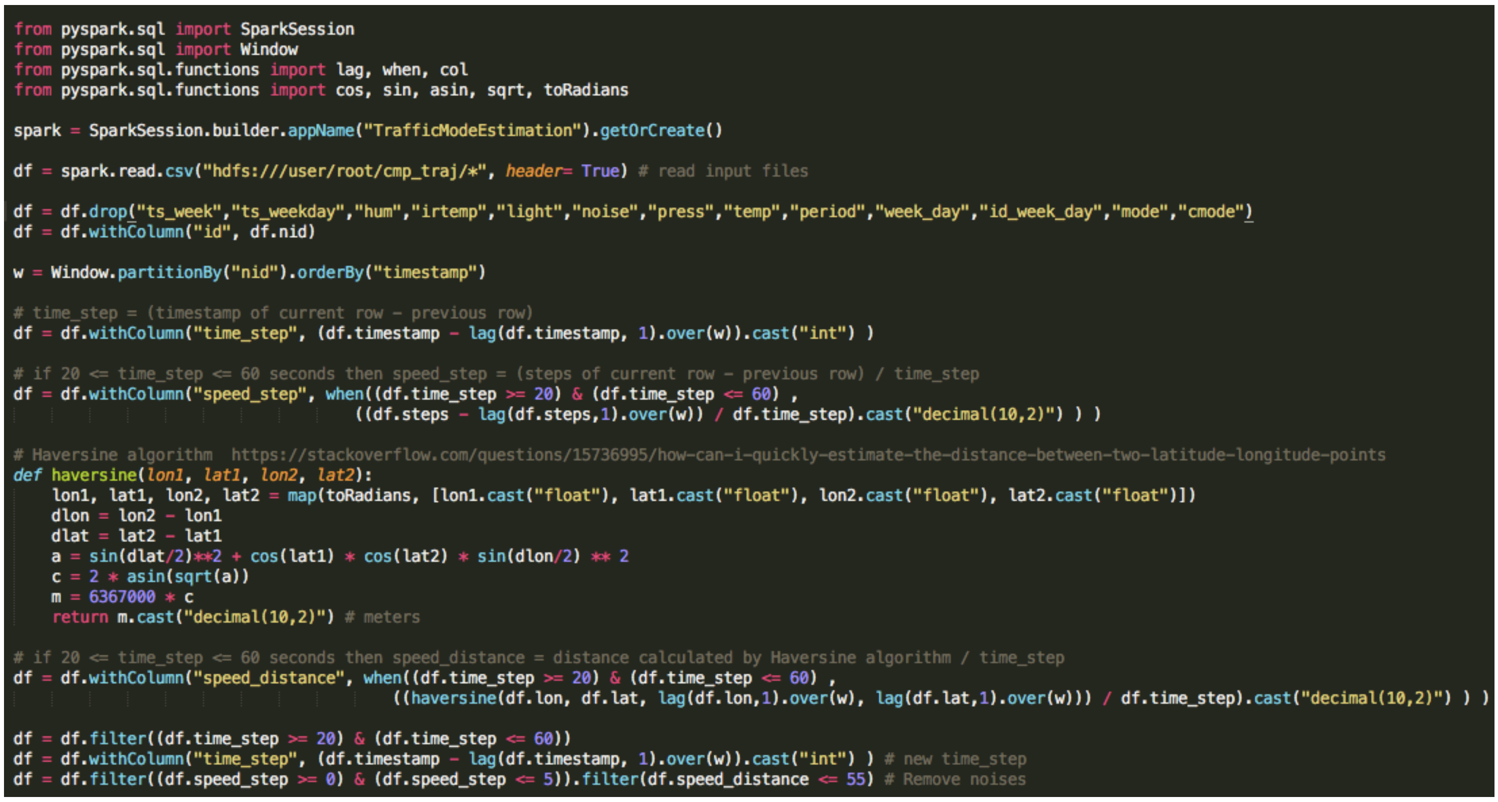


Figure 3. Result after remove noises

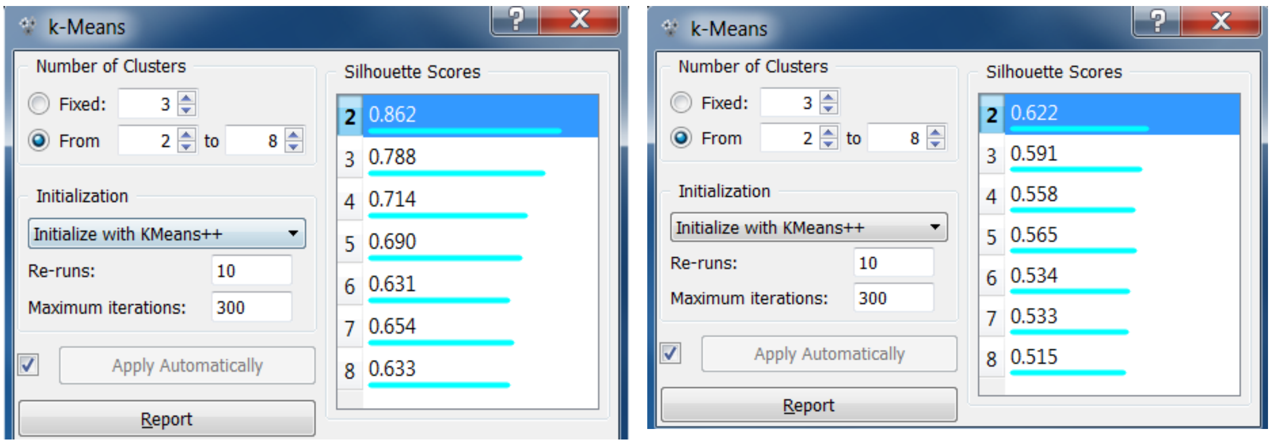
Here is the pyspark code to do the above task. It reads the csv files from HDFS, drop unused columns, add new column “time_step” which is calculated by timestamp of the current row minus timestamp of the previous row, add new column “speed_step” which is calculated by steps of the current row minus steps of the previous row, add new column “speed_distance” which is calculated by Haversine formula. The input of Haversine formula is the latitude and longitude of the current row and the previous row. After that, filter the record with time interval in 20-60 seconds, speed_step is non negative, and speed distance is not higher than 55.



```python
from pyspark.sql import SparkSession
from pyspark.sql import Window
from pyspark.sql.functions import lag, when, col
from pyspark.sql.functions import cos, sin, asin, sqrt, toRadians

spark = SparkSession.builder.appName("TrafficModeEstimation").getOrCreate()

df = spark.read.csv("hdfs:///user/root/cmp_traj/*", header= True) # read input files

df = df.drop("ts_week","ts_weekday","hum","irtemp","light","noise","press","temp","period","week_day","id_week_day","mode","cmode")
df = df.withColumn("id", df.nid)

w = Window.partitionBy("nid").orderBy("timestamp")

# time_step = (timestamp of current row - previous row)
df = df.withColumn("time_step", (df.timestamp - lag(df.timestamp, 1).over(w)).cast("int") )

# if 20 <= time_step <= 60 seconds then speed_step = (steps of current row - previous row) / time_step
df = df.withColumn("speed_step", when((df.time_step >= 20) & (df.time_step <= 60) ,
                                  ((df.steps - lag(df.steps,1).over(w)) / df.time_step).cast("decimal(10,2)") ) )

# Haversine algorithm  https://stackoverflow.com/questions/15736995/how-can-i-quickly-estimate-the-distance-between-two-latitude-longitude-points
def haversine(lon1, lat1, lon2, lat2):
    lon1, lat1, lon2, lat2 = map(toRadians, [lon1.cast("float"), lat1.cast("float"), lon2.cast("float"), lat2.cast("float")])
    dlon = lon2 - lon1
    dlat = lat2 - lat1
    a = sin(dlat/2)**2 + cos(lat1) * cos(lat2) * sin(dlon/2) ** 2
    c = 2 * asin(sqrt(a))
    m = 6367000 * c
    return m.cast("decimal(10,2)") # meters

# if 20 <= time_step <= 60 seconds then speed_distance = distance calculated by Haversine algorithm / time_step
df = df.withColumn("speed_distance", when((df.time_step >= 20) & (df.time_step <= 60) ,
                                      ((haversine(df.lon, df.lat, lag(df.lon,1).over(w), lag(df.lat,1).over(w))) / df.time_step).cast("decimal(10,2)") ) )

df = df.filter((df.time_step >= 20) & (df.time_step <= 60))
df = df.withColumn("time_step", (df.timestamp - lag(df.timestamp, 1).over(w)).cast("int") ) # new time_step
df = df.filter((df.speed_step >= 0) & (df.speed_step <= 5)).filter(df.speed_distance <= 55) # Remove noises
```

Traffic mode estimation

First, we use Orange (software for data mining) to do K-means clustering. We want 2 clusters (k = 2) and then do feature selection by silhouette score. With 2 features, speed_step (steps/sec), speed_distance (meter/sec), silhouette score is 0.86. With 3 features: acceleration, speed_step, speed_distance silhouette score is 0.622.


k-Means

Number of Clusters
Fixed: 3
[✓ From] 2 to 8

Initialization
Initialize with KMeans++
Re-runs: 10
Maximum iterations: 300

Apply Automatically
Report

Silhouette Scores

2	0.862
3	0.788
4	0.714
5	0.690
6	0.631
7	0.654
8	0.633

k-Means

Number of Clusters
Fixed: 3
[✓ From] 2 to 8

Initialization
Initialize with KMeans++
Re-runs: 10
Maximum iterations: 300

Apply Automatically
Report

Silhouette Scores

2	0.622
3	0.591
4	0.558
5	0.565
6	0.534
7	0.533
8	0.515

Figure 4: Silhouette score without acceleration(left) and with acceleration(right)

We can that see to classify traffic mode into 2 modes, it would be better to remove acceleration from the input features. Thus, the features will be speed_step and speed_distance.

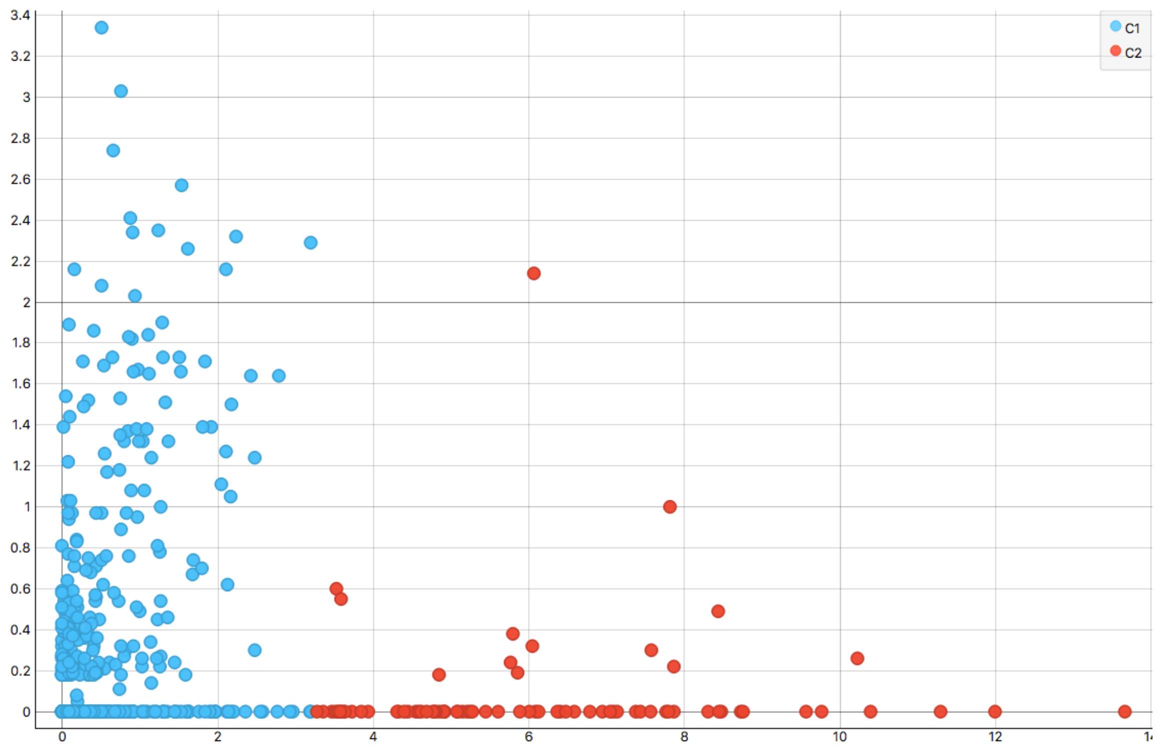


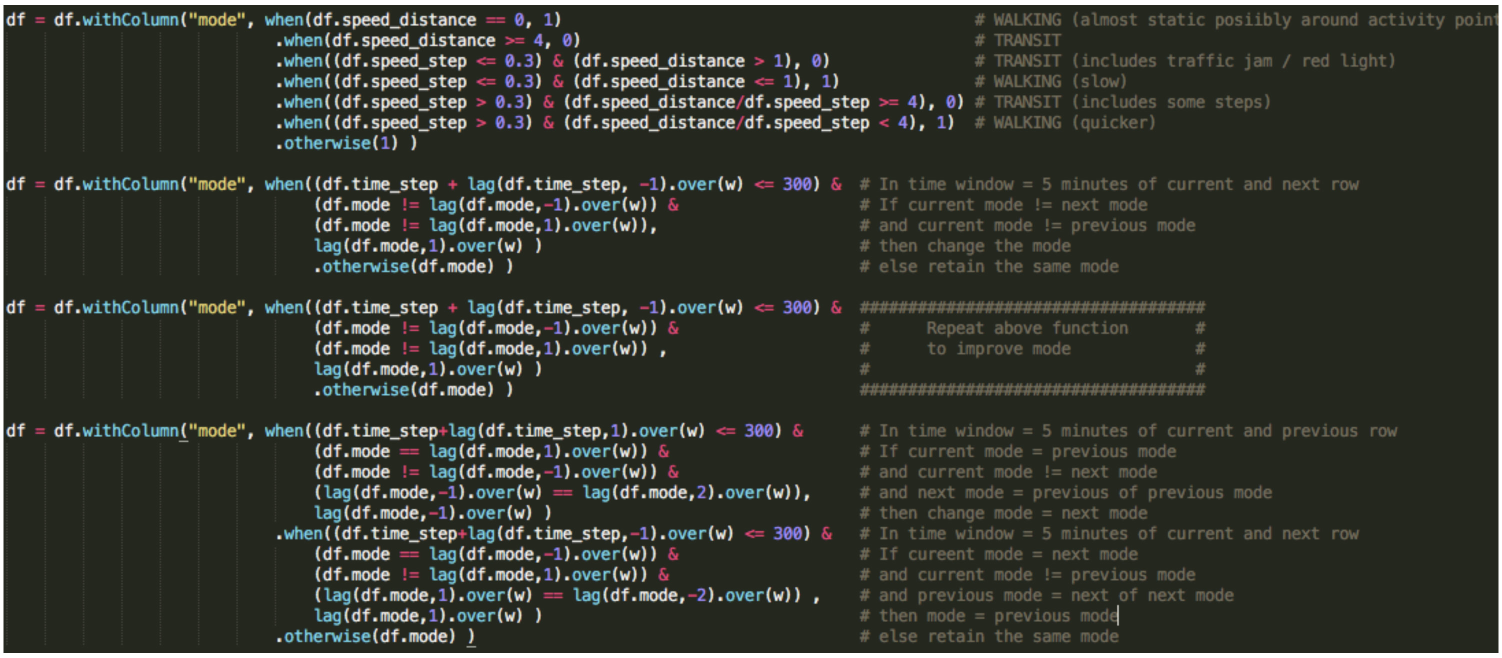
Figure 5: Clusters of speed distribution (a sample student)

We can observe the clusters of speed distribution here. Right now, let assume that blue points are walking and red points are transit according to the relationship between speed_step and speed_distance. We then set the rules to estimate the mode according to this relationship.

speed_step [steps/sec]	**speed_distance [m/s]**	**$\frac{\text{speed\_distance}}{\text{speed\_step}}$ ratio**	**Mode**
-	0	-	**WALKING**
-	>= 4	-	**TRANSIT**
<= 0.3	> 1	-	**TRANSIT**
<= 0.3	<= 1	-	**WALKING**
> 0.3	-	>= 4	**TRANSIT**
> 0.3	-	< 4	**WALKING**

Table 2: Rules for traffic mode estimation

Here is the code. The first section of code (lines1-7) is the rules to estimate the traffic mode. The other sections are the code to improve the traffic mode. The details are in the comment. We improve the traffic mode by changing the mode directly when it is not reasonable. For example, the mode changes from [walking] to [transit] to [walking] in a very short time interval.



```
df = df.withColumn("mode", when(df.speed_distance == 0, 1)                                      # WALKING (almost static posiibly around activity point
                              .when(df.speed_distance >= 4, 0)                                   # TRANSIT
                              .when((df.speed_step <= 0.3) & (df.speed_distance > 1), 0)         # TRANSIT (includes traffic jam / red light)
                              .when((df.speed_step <= 0.3) & (df.speed_distance <= 1), 1)        # WALKING (slow)
                              .when((df.speed_step > 0.3) & (df.speed_distance/df.speed_step >= 4), 0) # TRANSIT (includes some steps)
                              .when((df.speed_step > 0.3) & (df.speed_distance/df.speed_step < 4), 1)  # WALKING (quicker)
                              .otherwise(1) )

df = df.withColumn("mode", when((df.time_step + lag(df.time_step, -1).over(w) <= 300) &  # In time window = 5 minutes of current and next row
                                  (df.mode != lag(df.mode,-1).over(w)) &                 # If current mode != next mode
                                  (df.mode != lag(df.mode,1).over(w)),                   # and current mode != previous mode
                                  lag(df.mode,1).over(w) )                               # then change the mode
                                  .otherwise(df.mode) )                                  # else retain the same mode

df = df.withColumn("mode", when((df.time_step + lag(df.time_step, -1).over(w) <= 300) &  ####################################
                                  (df.mode != lag(df.mode,-1).over(w)) &                 #      Repeat above function       #
                                  (df.mode != lag(df.mode,1).over(w)) ,                  #      to improve mode             #
                                  lag(df.mode,1).over(w) )                               #                                  #
                                  .otherwise(df.mode) )                                  ####################################

df = df.withColumn("mode", when((df.time_step+lag(df.time_step,1).over(w) <= 300) &      # In time window = 5 minutes of current and previous row
                                  (df.mode == lag(df.mode,1).over(w)) &                  # If current mode = previous mode
                                  (df.mode != lag(df.mode,-1).over(w)) &                 # and current mode != next mode
                                  (lag(df.mode,-1).over(w) == lag(df.mode,2).over(w)),   # and next mode = previous of previous mode
                                  lag(df.mode,-1).over(w) )                              # then change mode = next mode
                              .when((df.time_step+lag(df.time_step,-1).over(w) <= 300) & # In time window = 5 minutes of current and next row
                                  (df.mode == lag(df.mode,-1).over(w)) &                 # If cureent mode = next mode
                                  (df.mode != lag(df.mode,1).over(w)) &                  # and current mode != previous mode
                                  (lag(df.mode,1).over(w) == lag(df.mode,-2).over(w)) ,  # and previous mode = next of next mode
                                  lag(df.mode,1).over(w) )                               # then mode = previous mode
                              .otherwise(df.mode) )                                      # else retain the same mode
```

We then plot it in the map to see the result by using Orange Geomap.

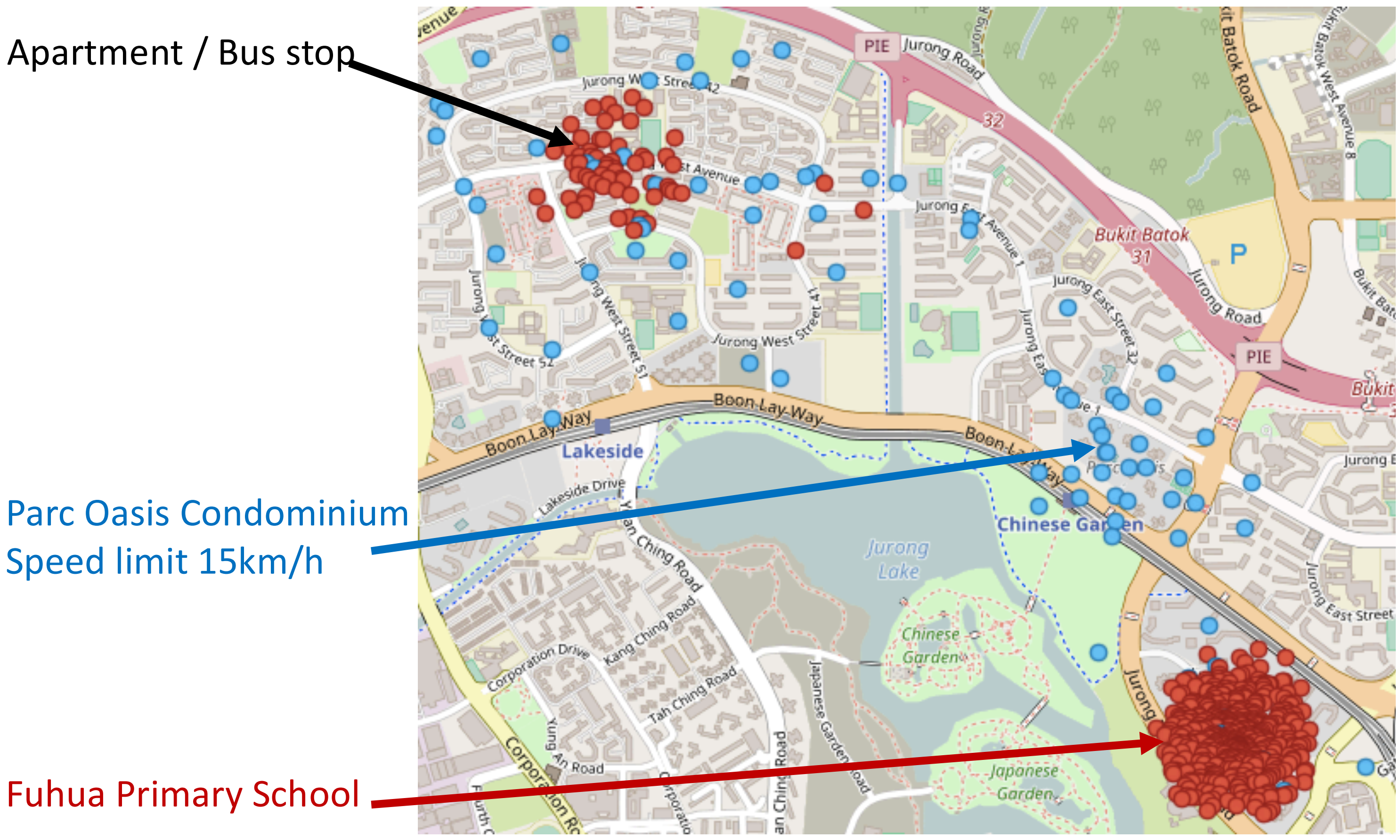


Figure 6: Points of a sample student

We can see from this figure that there are 2 main activity areas those are Apartment and School. The walking points (red points) should be around the activity area. The blue points are the path of transit from one activity area to the others. The result looks great as expected.

Activity area recognition

From the previous result, we want to aggregate the multiple walking points around activity area. The first approach does the following steps.

1. Find static points (speed = 0)
2. K-means clustering with features latitude, longitude, speed_step, speed_distance, time
3. Find centroid of each cluster

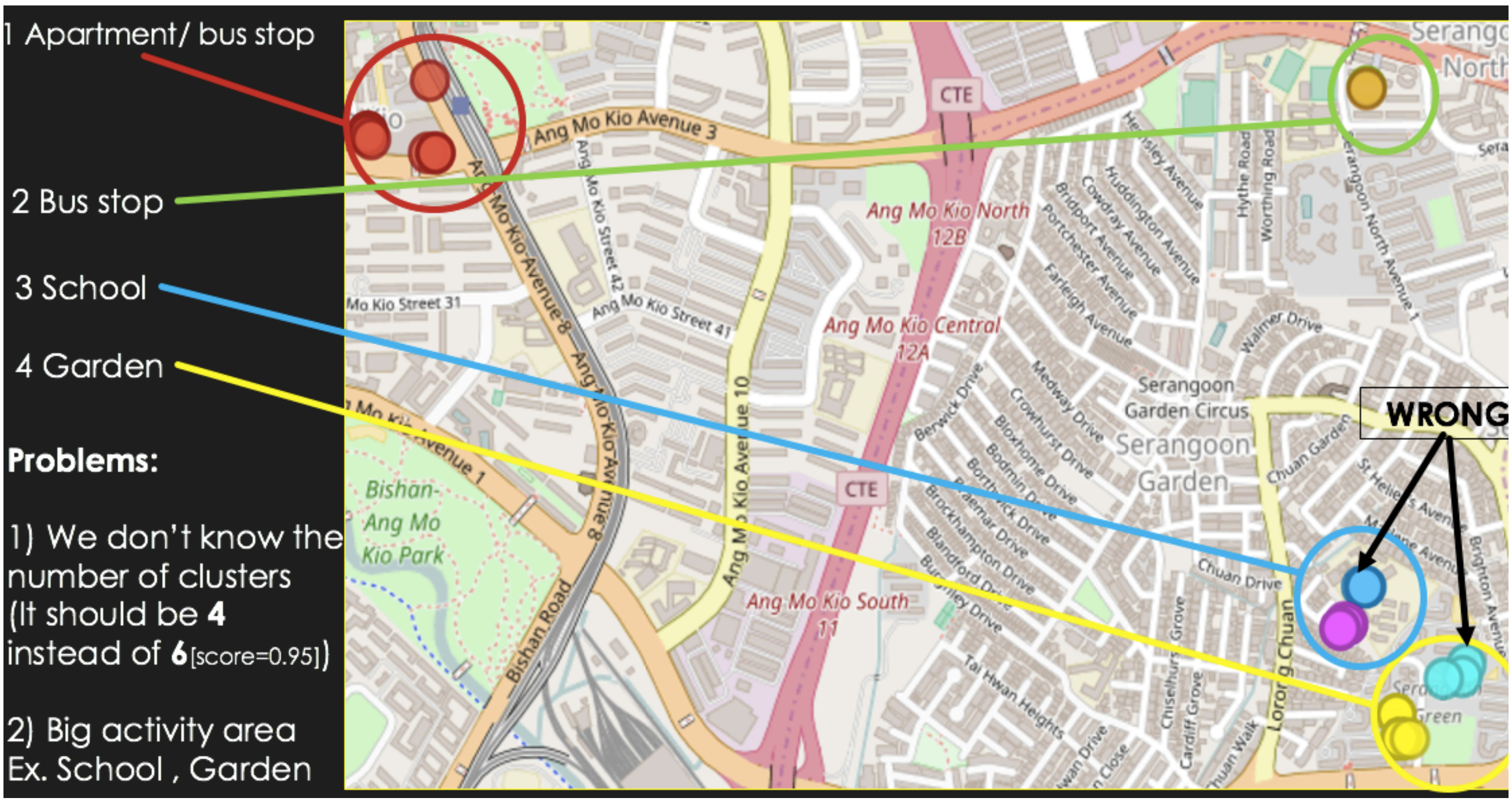
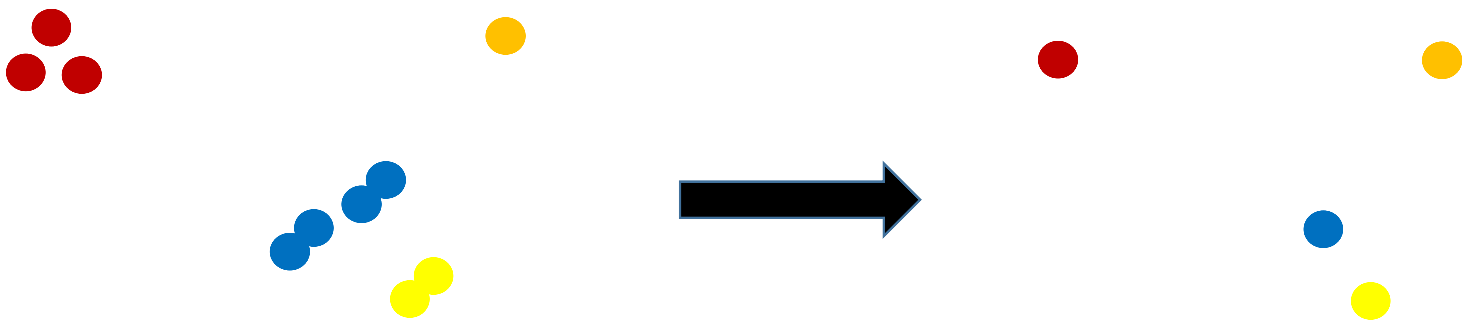

1 Apartment/ bus stop
2 Bus stop
3 School
4 Garden
Problems:
1) We don't know the number of clusters (It should be 4 instead of 6[score=0.95])
2) Big activity area Ex. School , Garden
WRONG

Figure 7: Result of the first approach of activity area recognition

Using clustering technique seems not working well. Even the silhouette score is high (0.95), the result is not correct because some activity areas are big. For instances, school and garden. Thus, we try the second approach.

The second approach does the following steps
1. Find static points (speed = 0)
2. Calculate distance between those points
3. If distance >= 300 meters then keep that point else remove the point.



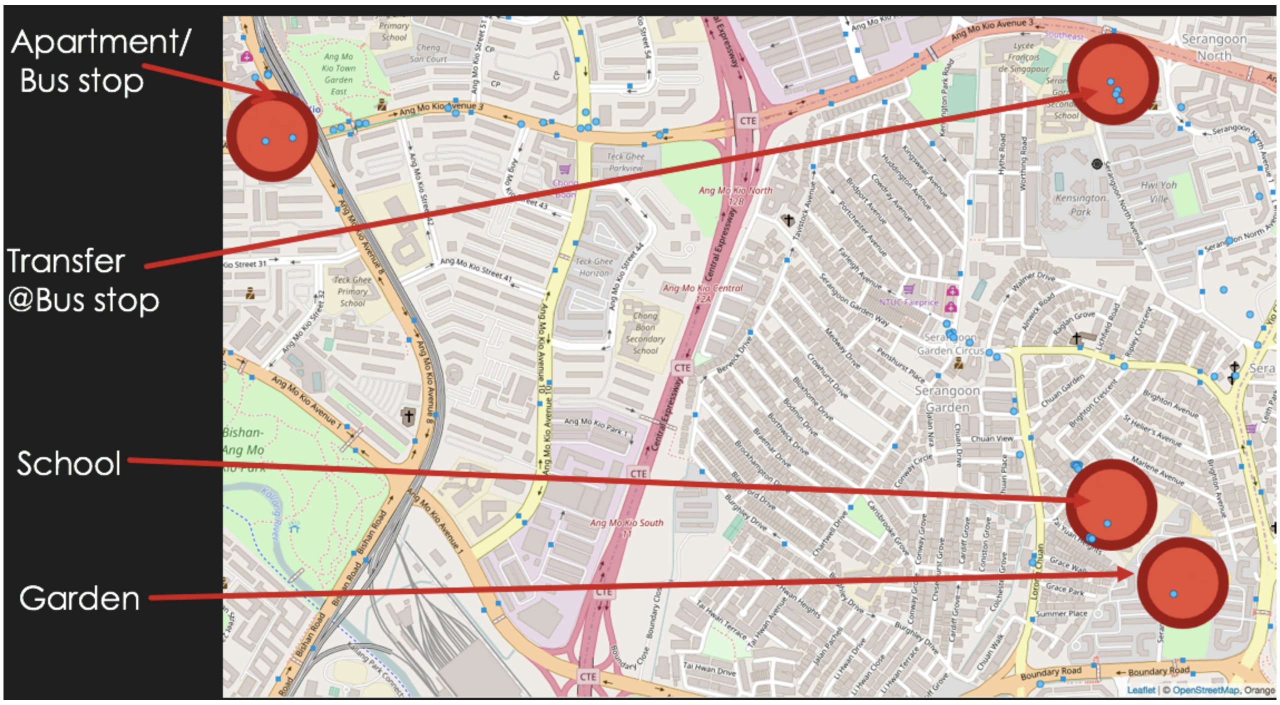


Figure 8: Result of the second approach of activity area recognition

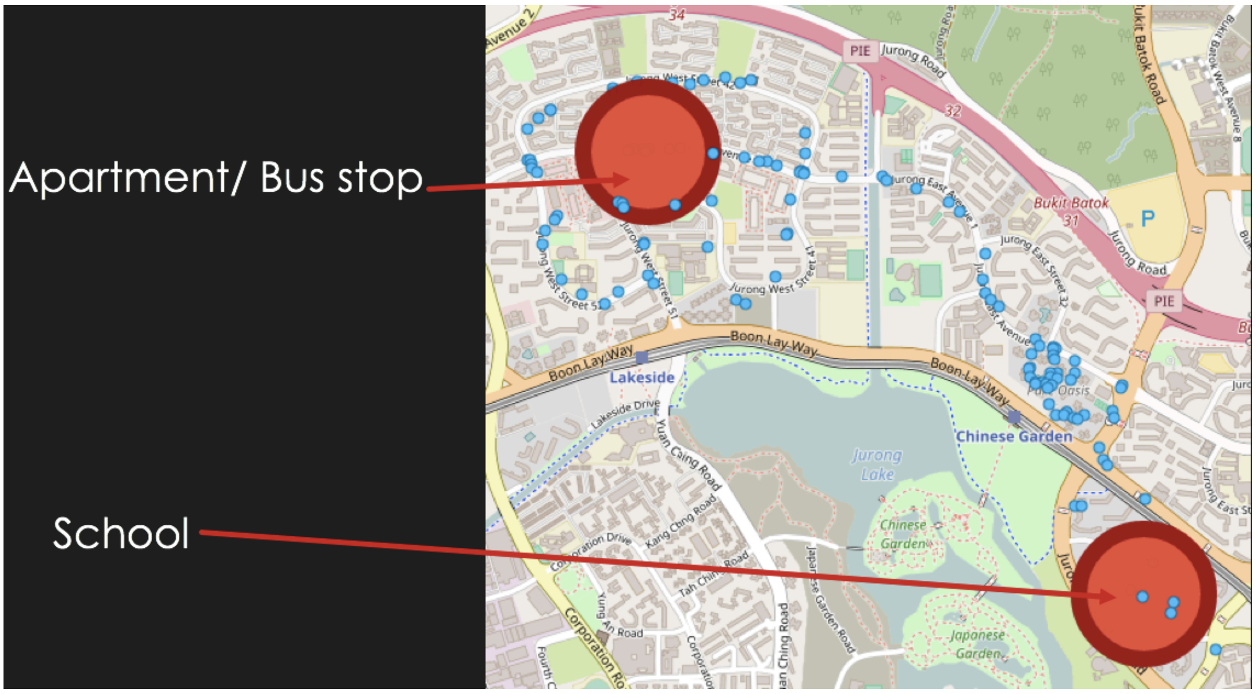
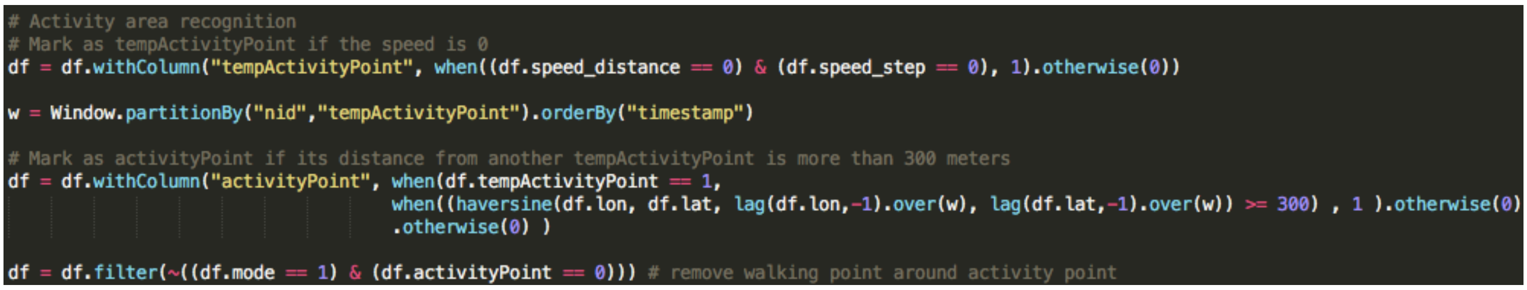


Figure 9: Another result of a student

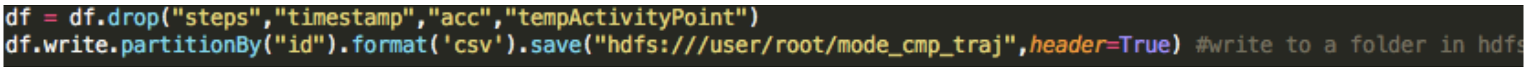
Now the results are correct and look great. The second approach is our solution.

Here is the code to do activity area recognition. The steps are already mention in the second approach.



```
# Activity area recognition
# Mark as tempActivityPoint if the speed is 0
df = df.withColumn("tempActivityPoint", when((df.speed_distance == 0) & (df.speed_step == 0), 1).otherwise(0))

w = Window.partitionBy("nid","tempActivityPoint").orderBy("timestamp")

# Mark as activityPoint if its distance from another tempActivityPoint is more than 300 meters
df = df.withColumn("activityPoint", when(df.tempActivityPoint == 1,
                                    when((haversine(df.lon, df.lat, lag(df.lon,-1).over(w), lag(df.lat,-1).over(w)) >= 300) , 1 ).otherwise(0)
                                    .otherwise(0) )

df = df.filter(~((df.mode == 1) & (df.activityPoint == 0))) # remove walking point around activity point
```

Finally, drop some columns and write csv file to hdfs



```
df = df.drop("steps","timestamp","acc","tempActivityPoint")
df.write.partitionBy("id").format('csv').save("hdfs:///user/root/mode_cmp_traj",header=True) #write to a folder in hdfs
```

Interactive queries

You can run the script locally on a single by using command **pyspark** to enter the pyspark shell.

Submit the job to the cluster

spark-submit --master spark://master:7077 –executor-memory 9G mode_estimation.py

You can tune the number of memory by --executor-memory <u>x</u>G

References

[1] http://mpi4py.readthedocs.io/en/stable/

[2] https://en.wikipedia.org/wiki/OpenMP

[3] https://en.wikipedia.org/wiki/CUDA

[4] https://distributed.readthedocs.io/en/latest/

[5] https://www.tutorialspoint.com/hadoop/index.htm

[6] https://www.tutorialspoint.com/apache_spark/index.html

[7] https://www.researchgate.net/post/Which_parallelising_technique_OpenMP_MPI_CUDA_would_you_prefer_more

[8] https://stackoverflow.com/questions/1530490/what-are-some-scenarios-for-which-mpi-is-a-better-fit-than-mapreduce?rq=1

[9] http://dask.pydata.org/en/latest/examples/bag-word-count-hdfs.html

[10] http://www.michael-noll.com/tutorials/writing-an-hadoop-mapreduce-program-in-python/

[11] https://stackoverflow.com/questions/37871194/how-to-tune-spark-executor-number-cores-and-executor-memory

[12] https://spark.apache.org/docs/latest/