CODE BOOK

Code book provides information about different tools, modules, and scripts used to analyse twenty-five journey data files.

Requirements

1. The program was run on below specification
   - Operating System: *Windows 64-bit Operating System*
   - Development Environment: *Spyder 3.3.6*
   - Language: *Python 3.7*
   - *Web Application: Jupyter Notebook 6.0.1*

2. Packages with version where used to run the program
   - pip V 19.0.3
   - seaborn V 0.9.0
   - plotly V 4.5.1
   - pandas V 1.0.1 (at least V 0.25+ is required)
   - altair V 4.0.1
   - numpy V 1.16.2
   - matplotlib V 3.0.3
   - os (latest)
   - datetime (latest)
   - math (latest)

Python Files

Three, (dot).py files were used to analyse journeys' data.

a. built_data.py: This file contains three user-defined functions.
   i. filebrowser()
   ii. read_telematic()
   iii. cal_dist_dur()

The filebrowser() is used to extract all the journeys' .csv files from the provided directory. *This function serves to extract all the CSV files.*

The read_telematic() is used to read the csv's (one-at-a-time), allocate session ID (based on file name), allocate journey ID (unique ID for each journey, e.g. JID2), conversion of Unix Epoch Time, appending datetime object, save all journeys data in one csv file and return combined dataframe. *This function serves to (based on files' location) combine and append new columns to one data frame.* Refer to **journeys.csv** (under *data* folder) for this function's results.

The cal_dis_dur() is used to calculate distance (in miles), and duration (in minutes) based on longitude and latitude information, by filtering source type as 'gps' and for single journey file only. Refer to **singlejourney.csv** (under *data* folder) for this function's results.

b. sensor.py: This file contains nine user-defined functions.
   i. event_accelerometer()
   ii. sensor_dist()
   iii. vel_speed()

iv. cor_speed_axes()
v. cor_speed_height()
vi. axes_measures()
vii. gps_measures()
viii. visual_accelerometer()
ix. visual_gps()

The event_accelerometer() is used to provide interactive visualisation about accelerometer axes (x, y, & z) against time registered during single journey. Refer to event_accelerometer.html file under output folder for result.

The sensor_dist() is used to elicit the numerical data distribution about accelerometer's axes (x, y, &z). It gives information about measurement of different axes in relationship to it centrality. Refer to sensor_dist.png file under output folder for result.

The vel_speed() is used to elicit inform about vehicle speed in meter per second over time. Refer to vel_speed.html file under output folder for result.

The cor_speed_axes() is used to inform about correlation between speed and accelerometer axes (x, y, & z). This operation also elicit visualisation. Refer to cor_speed_axes.png & cor_speed_axes.csv files under output folder for result.

The cor_speed_height() is used to inform about correlation between speed and height registered during the journey. This operation also elicit visualisation. Refer to cor_speed_height.png & cor_speed_height.csv files under output folder for result.

The axes_measures() is used to provide dispersion & centrality measures about accelerometer axes (x, y, & z). Refer to axes_measures.csv file under output folder for result.

The gps_measures() is used to provide dispersion & centrality measures about speed, accuracy, height. Refer to gps_measures.csv file under output folder for result.

The visual_accelerometer() is used to provide visualisation about axes' standard deviation in shades . Refer to visual_accelerometer_std_x.png, visual_accelerometer_std_y.png, & visual_accelerometer_std_z.png files under output folder for result.

The visual_gps() is used to provide visualisation about axes' standard deviation in shades . Refer to visual_gps_std_speed.png, visual_gps_std_accuracy.png, & visual_gps_std_height.png files under output folder for result.

c. event_trigger.py: This file contains two user-defined functions.
   i. event_teasing()
   ii. visual_event()

The event_teasing() is used to identifies, and allocates *notional* 'severity index' & 'confidence' about the trip containing an 'event' that might be an accident. It also generates event_teasing.csv file. Refer to event_teasing.csv file under output folder for result.

The visual_event() is used to visualise the notional confidence along speed over time. Refer to visual_event.html file under output folder for result.

<u>Execution Details</u>

- File built_data.py is required to be executed first as below.
  - Execute all import statements (with version as mentioned above).
  - Run the **BuiltData** class.
  - Using unit test cases validate the class by creating class object and execute its function's.
    1. BuiltData() requires directory path where all the 25 journey files are located.
    2. filebrowser() can be executed without need of argument.
    3. read_telematic() requires List object returned by filebrowser().
    4. cal_dist_dur() requires dataframe returned by read_telematic(), source type as 'gps' only, and any one journey ID between 'JID1' to 'JID25'. Note **JID** should be capital, in quotes.

- Next, sensor.py file must be executed.
  - Execute all import statements (with version as mentioned above).
  - Run the **Sensor** class.
  - Using unit test cases validate the class by creating class object and execute its function's.
    1. Sensor() requires directory path of journeys.csv file (that was created through read_telematic() function of BuiltData class).
    2. event_accelerometer(arg1, arg2, arg3) requires sensor.directory as first argument, any one journey ID between 'JID1' to 'JID25' (Note **JID** should be capital, in quotes) as second argument and '*accelerometer' (in quotes)* as last argument.
    3. sensor_dist(arg1, arg2, arg3) requires *sensor.directory* as first argument, any one journey ID between 'JID1' to 'JID25' (Note **JID** should be capital, in quotes) as second argument and '*accelerometer' (in quotes)* as last argument.
    4. vel_speed(arg1, arg2, arg3) requires *sensor.directory* as first argument, any one journey ID between 'JID1' to 'JID25' (Note **JID** should be capital, in quotes*)* as second argument and '*gps' (in quotes)* as last argument.
    5. cor_speed_axes(arg1, arg2) requires *sensor.directory* as first argument, any one journey ID between 'JID1' to 'JID25' (Note **JID** should be capital, in quotes) as last.
    6. cor_speed_height(arg1, arg2) requires *sensor.directory* as first argument, any one journey ID between 'JID1' to 'JID25' (Note **JID** should be capital, in quotes) as last.
    7. axes_measures(arg1) requires *sensor.directory* as argument.
    8. gps_measures(arg1) requires *sensor.directory* as argument.
    9. visual_accelerometer(arg1) requires *sensor.directory* as argument.
    10. visual_gps(arg1) requires *sensor.directory* as argument.

- Lastly, event_teasing.py file must be executed. This file takes couple of minutes to execute.
  - Execute all import statements (with version as mentioned above).
  - Run the **EventTrigger** class.
  - Using unit test cases validate the class by creating class object and execute its function's.
    1. EventTrigger() requires directory path of journeys.csv file (that was created through read_telematic() function of BuiltData class).
    2. event_teasing(arg1) requires event.directory as an argument.
    3. visual_event(arg1, arg2) requires dataframe object (returned by event_teasing function) as first argument, and any one journey ID between 'JID1' to 'JID25' (Note **JID** should be capital, in quotes) as last argument.

Jupyter Notebook

TheFloow.ipynb file contain single functionality to examine the speed & distance covered over time. Using plotly feature an interactive visualisation of each journey information has been elicited.

Execution Details

- File TheFloow.ipynb requires directory path of ‘singlejourney.csv’ file (generated through built_data.py)
- Execute each cell at-a-time using either CTLR + ENTER or Run button (seen on the Jupyter notebook)