GSR Spring 2019

Final Report

(Dr Jane Macfarlane)

Keshav Kothari

TABLE OF CONTENTS:

1. Visualization

The first task for the semester was to effectively visualize the traffic data to convey important parameters like region of congestions across a large area without it being too overwhelming for the user given the size of the area. Deck.gl by uber was chosen as the tool to implement these visualizations. Deck.gl is a WebGL-powered framework for visual exploratory data analysis of large datasets. Deck.gl offers an extensive catalog of pre-packaged visualization "layers", including ScatterplotLayer, ArcLayer, TextLayer, GeoJsonLayer, etc. The input to a layer is usually an array of JSON objects. Each layer offers a highly-flexible API to customize how the data should be rendered.

A significant amount of time was spent understanding the library and get it working. Deck.gl works with Mapbox maps and necessary authentication for that is required to load maps on the website which is important in the current scenario. The GitHub repository for deck.gl contains a lot of example code, which was used to understand how different layers function. The layers that were relevant to visualize trips were the trips and geojson layers.

To clone the repository and run an example file, follow the given steps:

```
git clone git@github.com:uber/deck.gl.git

cd deck.gl/examples/get-started/pure-js/basic

npm install

export MapboxAccessToken={Your Token Here} && npm start
```

1.1 Trips layer

The trips layer can visualize individual trips. It takes as an input a list of json objects, one for each vehicle, containing an attribute **segments**, which is a list of coordinates and timestamps of that individual vehicle. The *app.js* file has a time counter and displays the relevant coordinates based on the current time in the code. The trips can be given as a path to *trips.json* file.

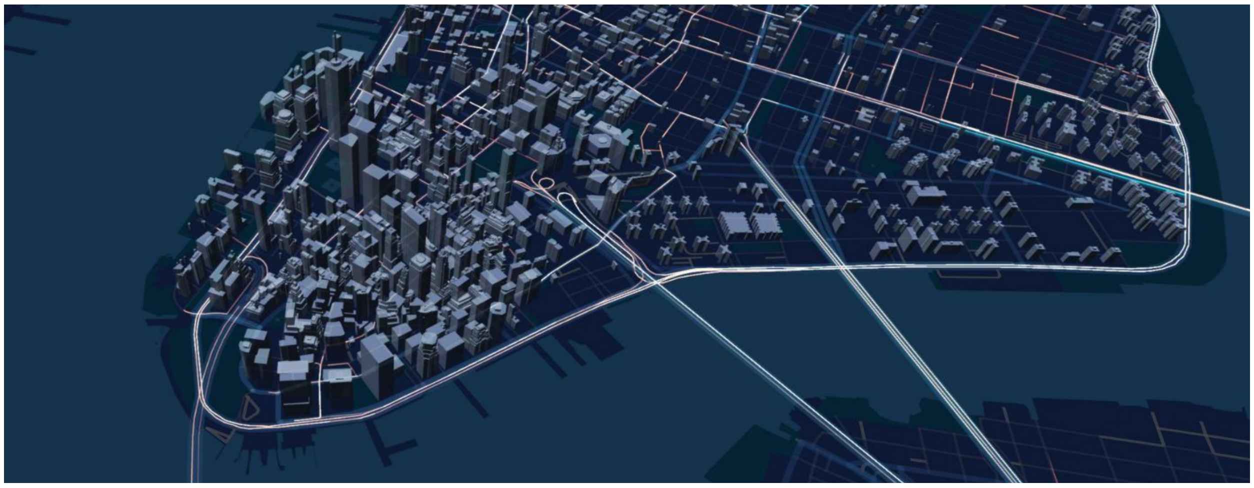
```
[{

  "vendor": 0,

  "segments": [
    [-73.78966, 40.6429, 1338],

    [-73.7895, 40.64267, 1346.233]

}]
```

The above is an example of a *trips.json* file which has one vehicle with id ‘0’ and has spatial coordinates for two timestamps. The code also accepts a path to *building.json* file which has the necessary details for render buildings and makes the visualization more realistic.

```
[{

  "height": 137,

  "polygon": [

    [-74.01766, 40.70552],

    [-74.01752, 40.70572],

    [-74.01766, 40.70552]

  ]

}]
```

The above is an example of a *buildings.json* file. The file contains a list of objects, each of which has information to render one building. The object has an attribute height which is the height of the building and a polygon shapefile which has spatial coordinates of the building as a polygon. The polygon must be a closed polygon, i.e, it must start and end on the same coordinates.

As mentioned above, a mapbox token is required to render the map. Features like initial view, loop length, animation speed, and trail lengths are variable and can be adjusted based on requirements. The map style can also be adjusted.



The figure above represents an example of the trip layer visualization. It is useful when the number of vehicles is less but as the number of vehicles increases, it becomes increasingly difficult to view important details on the map. The visualization is also computationally expensive and an increase in the number of vehicles slows down the animation drastically.

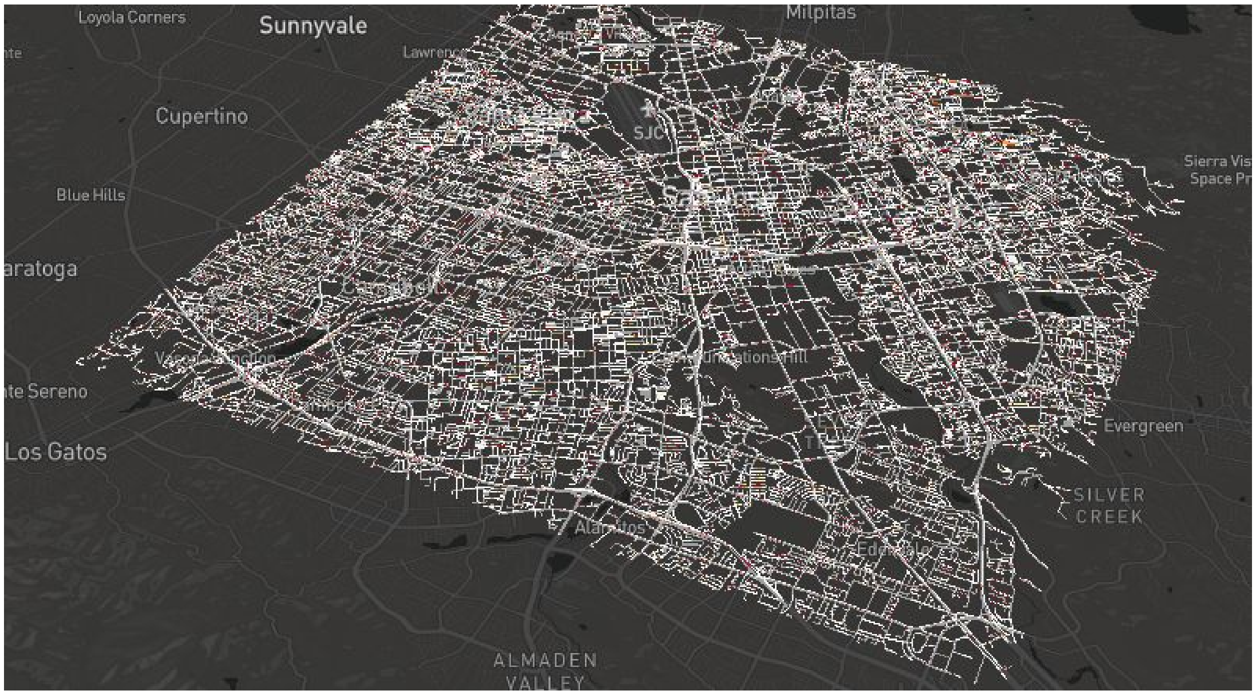
1.2 GeoJSON layer

The geojson layer can visualize geojson objects on the map. As with the trips layer, to visualize maps, a mapbox authentication is required. The geojson layer was used because visualizing individual trips for such a large region was computationally expensive. It also isn't useful in visually identifying areas of congestions. The geojson layer inputs a geojson file which is of the format mentioned below.

```
{"type": "FeatureCollection",

"features": [

{"type":  "Feature",  "properties":  {  "jcolor":  "  #FFC000"},"geometry":  {"type": "LineString", "coordinates": [[-121.83581, 37.32305], [-121.8388, 37.32614]]}},

{"type":  "Feature",  "properties":  {  "jcolor":  "  #FFFF00"},"geometry":  {"type": "LineString", "coordinates": [[-121.8388, 37.32614], [-121.83924, 37.32661]]}}

]}
```

The json object is of the type feature collection, with the feature attribute containing a list of features, each of which is of the type '**LineString**'. Each feature in the list represents a link (road segment), with the coordianates of the two nodes and a jcolor attribute that corresponds has a color code corresponding to heatmap that tells the flow/capacity ratio for the link. The file then has the same information for all the links in the network and hence can visualize traffic on a much larger scale, along with being able to clearly mark areas that might be congestion prone.

The image below is an example of the visualization from using the geojson layer for links in the San Jose area in California, US. As can be seen from the image, it is easy to identify regions that have high traffic congestion. The next step to this layer was to animate these geojson features with time to visualize the change across the day. This process, however, was difficult because of the lack of documentation. The problem was solved once kepler.gl was identified which uses the deck.gl backend but has numerous front end features which makes it user-friendly. The data file uploaded on the kepler.gl website is automatically processed if the columns are labeled correctly and it suggests the layers that it perceives would be the best. Hence, kepler.gl solved the problem of visualization and no more work was done on animating the geojson layer.

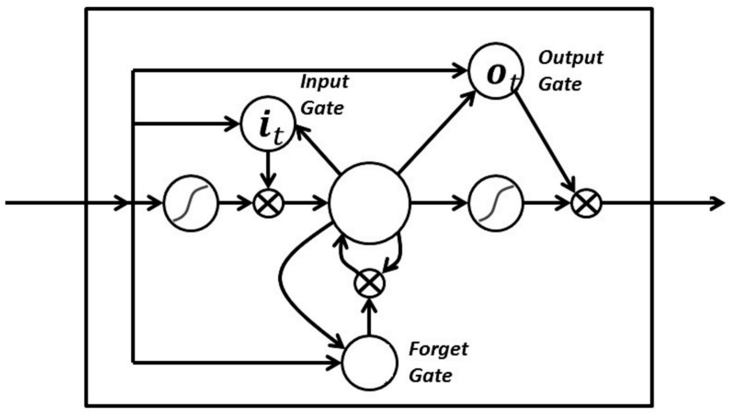


2. Maneuver Classification using RNN LSTM

The next task was to identify maneuvers for individual vehicle probe data (i.e, a collection of consecutive coordinates for a vehicle). Maneuvers are an important feature in both delay calculations and fuel consumption calculations. Vehicles tend to consume more fuel when they're taking a right turn or a left turn as compared to going straight at an intersection. Similarly, left turns are usually more expensive than right turns (one of the reasons why UPS delivery trucks are routed in a way that minimizes the number of left turns).

A neural network approach was opted for solving this problem. Recurrent neural networks are very effective in solving problems where inputs are a time series, which is so in our case. The input to the network was the relative x and y co-ordinates of the vehicles, sampled at a constant interval of 5 seconds. Each input had data 12 data points, corresponding to 1 min worth of data. The assumption here was that each input data had at most one maneuver and the time step was chosen accordingly. Since the real data rarely has a constant sampling rate, the data was extrapolated to obtain coordinates corresponding to the required time steps. Also, as mentioned above, the spatial data for each coordinate was normalized wrt to the first point, which meant that the first point always had [0,0] as coordinates.

Long Short-Term Memory (LSTM) networks are an extension for recurrent neural networks, which basically extends their memory. Therefore it is well suited to learn from important experiences that have very long time lags in between. The units of an LSTM are used as building units for the layers of an RNN, which is then often called an LSTM network. LSTM's enable RNN's to remember their inputs over a long period of time. This is because LSTM's contain their information in a memory, that is much like the memory of a computer because the LSTM can read, write and delete information from its memory. This memory can be seen as a gated cell, where gated means that the cell decides whether or not to store or delete information (e.g if it opens the gates or not), based on the importance it assigns to the information. The assigning of importance happens through weights, which are also learned by the algorithm. This simply means that it learns over time which information is important and which not. In an LSTM you have three gates: input, forget and output gate. These gates determine whether or not to let new input in (input gate), delete the information because it isn't important (forget gate) or to let it impact the output at the current time step (output gate). The figure below represents an illustration of an RNN with its three gates.



The RNN was implemented using the tensorflow library, which can be installed and imported using the following commands.

```
pip install tensorflow

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, Dropout, LSTM
```

2.1 Generate Training Data

The primary problem with using a neural network problem is to generate training data. Labeling individual trajectories for their maneuvers is a labor-intensive process and consumes a lot of time. Hence, a script was written to autogenerate trajectories, which the neural network can be then trained on. To generate realistic trajectories, the following features were added.

- Each trajectory had 12 points, corresponding to 1 minutes worth of data
- The vehicles can start with any heading, chosen randomly
- The vehicles could turn at any of the 12 points in any direction (left, right or U-turn) with a probability value of 1/48, which makes all the four maneuvers (straight, left, right or U-turn) equally likely, giving a balanced training dataset
- Each trajectory has at most one maneuver, in line with the assumption made above
- The vehicles start with a random velocity and accelerate and deaccelerate randomly within a certain threshold if no maneuvers are made
- The vehicles slow down to make a maneuver to a speed of 15 mph, using realistic acceleration rates
- Error is induced in the position recorded for the vehicles by displacing the point anywhere within a circle based on the average GPS errors
- Error is induced in the speeds recorded for the vehicles

These features are used to generate 60000 data points for training, along with 15000 data points each for cross-validation and testing respectively.

2.2 Model Framework

The neural network is a sequential network with 2 LSTM layers of size 128 and 1 Dense layer of size 32. The final layer was a dense layer of size 4, corresponding to the number of output classes. The hidden layers also have dropouts with 0.2 probability.

```
model = Sequential()

model.add(LSTM(128,input_shape=(x_train.shape[1:]),activation='relu',return_sequences=
True))

model.add(Dropout(0.2))

model.add(LSTM(128,activation='relu'))

model.add(Dropout(0.2))

model.add(Dense(32,activation='relu'))

model.add(Dropout(0.2))

model.add(Dense(4,activation='softmax'))
```

Adams optimizer was used with learning rate = 1e-3 and decay = 1e-5.

```
opt = tf.keras.optimizers.Adam(lr=1e-3, decay=1e-5)
```

```
model.compile(loss='sparse_categorical_crossentropy',
              optimizer=opt,
              metrics=['accuracy'])
model.fit(x_train,y_train,epochs=3,validation_data=(x_val,y_val))
```

*Note: The above mentioned framework is the result of tuning hyperparameters. Several other parameters were tried and this had the best validation accuracy.

2.3 Testing

The neural network on testing on test dataset after training for 3 epochs had an accuracy of **95.24%**, which was a fair point to start with. The model was then tested on real datasets, preprocessed to a sampling rate of 5 seconds. The resulting accuracy was **87.32%**.

Although the model was fairly accurate in predicting the maneuver, it was not trained to predict the point of maneuver and corresponding features like the radius of curvature of the turn, which we realized were important in accurately determining fuel consumptions. Hence, a more mathematical approach was used, as mentioned in the section below.

3. Turning Point Identification & Classification

A neural network is an empirical approach to solving a problem where we train a model on a dataset. However, to identify turning points, we recognized a need to have a more analytical approach. The analytical approach is based on creating vectors out of every consecutive data points and then calculating the angle between those vectors using dot product.

3.1 Turning Point Identification

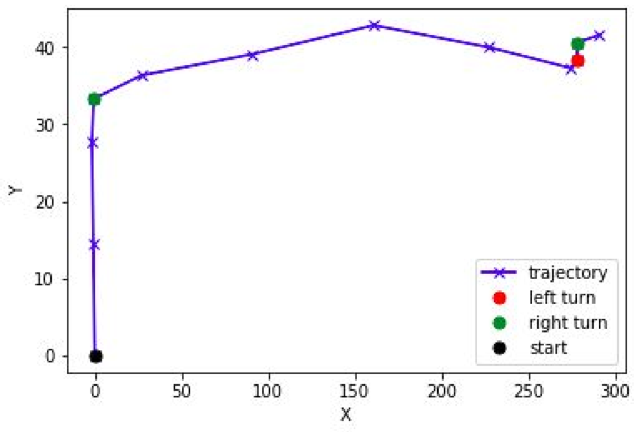
As mentioned above, vectors were calculated for a sequence of data points and the angle between consecutive vectors was calculated using dot products. A point was classified as a turning point if the angle between the two vectors was greater than 45°.

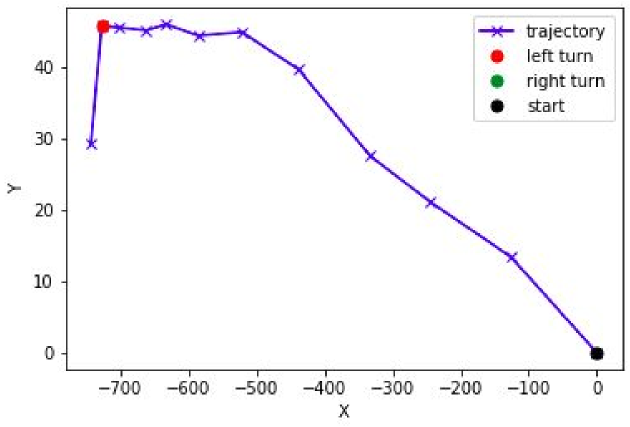
3.2 Turning Point Classification

A point is classified as a right turning point if the vector B is clockwise of vector A, where vector A is the vector for the t-1 timestep and vector B is the vector for the tth time step.

```
# (B is clockwise of A) if x v > y u => right turn
# A=(x,y), B=(u,v)

# A -> t-1, B -> t (timesteps)
```

The classification was made on the formula indicated below. From the series of all vectors, ids of points were noted where the angle between them was greater than 45°. Then if the vector B was clockwise of vector A, it was added to the list of ids for right turning maneuver, ow, it was added to the list of left turning maneuver. The maneuver was classified as a U-turn if the angle was greater than 150°. All of the values were obtained from the datasets.





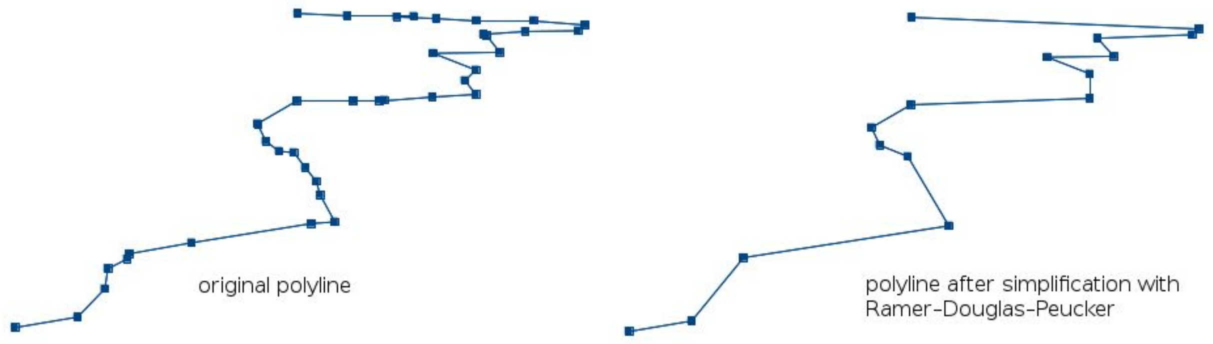
The images above represent a couple of example of turning points identification and classification using this analytical method. It overall had worse accuracy as compared to the RNN method in turning point classification but is more robust with trajectories and can identify the point of the turn, which is difficult using an RNN.

3.3 Trajectory Smoothing

The above approach had a couple of issues, mainly dealing with the way data was obtained. The first was that if the data was too erroneous, points could be identified as turning even on straight roads because of the zig-zag pattern in the data. The resolution to this was trajectory smoothing which aims to reduce the noise in trajectories and avoid misclassification of trajectories.

The Ramer–Douglas–Peucker algorithm, also known as the Douglas–Peucker algorithm and iterative end-point fit algorithm, is an algorithm that decimates a curve composed of line segments to a similar curve with fewer points. The algorithm defines 'dissimilar' based on the maximum distance between the original curve and the simplified curve (i.e., the Hausdorff distance between the curves). The simplified curve consists of a subset of the points that defined the original curve.

The starting curve is an ordered set of points or lines and the distance dimension $\varepsilon > 0$. The algorithm recursively divides the line. Initially, it is given all the points between the first and last point. It automatically marks the first and last point to be kept. It then finds the point that is furthest from the line segment with the first and last points as endpoints; this point is obviously furthest on the curve from the approximating line segment between the end points. If the point is closer than ε to the line segment, then any points not currently marked to be kept can be discarded without the simplified curve being worse than ε. If the point furthest from the line segment is greater than ε from the approximation then that point must be kept. The algorithm recursively calls itself with the first point and the furthest point and then with the furthest point and the last point, which includes the furthest point being marked as kept. When the recursion is completed a new output curve can be generated consisting of all and only those points that have been marked as kept. The figure below indicates a simple example of using an RDP.



The RDP was implemented using the rdp library, which can be installed, imported and used using the following commands.

```
pip install rdp
```

```
from rdp import rdp
```

```
simplified_trajectory = rdp(trajectory, epsilon=0.002)
```

Similarly, other filters like Kalman filters can be used to smoothen the trajectories. Subsampling can also be used to smoothen the trajectories, a methodology for which is suggested in the paper **Temporal Sampling Constraints for GeoSpatial Path Reconstruction in a Transportation Network** by *Jane Macfarlen* and *Bo Xu*. The paper suggests a minimum sampling interval to accurately recreate routes from probe data. However, subsampling still leaves behind the issue where the data is infrequent and thus has no information about the intersection at all. Under such circumstances, information about the road networks is essential in determining the route taken by the user. Further, road networks are marked fairly accurately and matching a trajectory on the road network solves all issues involving lateral GPS errors associated with the data.

4. OSRM Map Matching

The Open Source Routing Machine or OSRM is a C++ implementation of a high-performance routing engine for shortest paths in road networks. It runs on OpenStreetMap data and has the following services available via HTTP API, C++ library interface and NodeJs wrapper.

- Nearest - Snaps coordinates to the street network and returns the nearest matches
- Route - Finds the fastest route between coordinates
- Table - Computes the duration or distances of the fastest route between all pairs of supplied coordinates
- Match - Snaps noisy GPS traces to the road network in the most plausible way
- Trip - Solves the Traveling Salesman Problem using a greedy heuristic
- Tile - Generates Mapbox Vector Tiles with internal routing metadata

The match feature was used to snap the trajectory to a road network and then use those coordinates for the analytical algorithm mentioned above. The OSMR can be installed and used following the steps mentioned below.

```
apt install docker.io
```

```
docker pull osrm/osrm-backend
```

```
wget https://download.geofabrik.de/north-america/us/california/socal.html
```

The link above downloads the OSM map for the southern California region, including the LA region, as represented in the figure below.

Salinas
California
Visalia
Las Vegas
Bakersfield
Santa Maria
Lancaster
Oxnard
Santa Clarita
San Bernardino
Los Angeles
Menifee
Oceanside
Tijuana
Yuma
Mexicali
Ensenada
Baja California

```
docker run -t -v "${PWD}:/data" osrm/osrm-backend osrm-extract -p /opt/car.lua /data/socal-latest.osm.pbf
```

```
docker run -t -v "${PWD}:/data" osrm/osrm-backend osrm-partition /data/socal-latest.osrm
```

```
docker run -t -v "${PWD}:/data" osrm/osrm-backend osrm-customize /data/socal-latest.osrm
```

The three commands mentioned above are used to process the osm data and index it in a format that allows for quick geospatial queries.

```
docker run -t -i -p 5000:5000 -v "${PWD}:/data" osrm/osrm-backend osrm-routed --algorithm mld /data/socal-latest.osrm
```

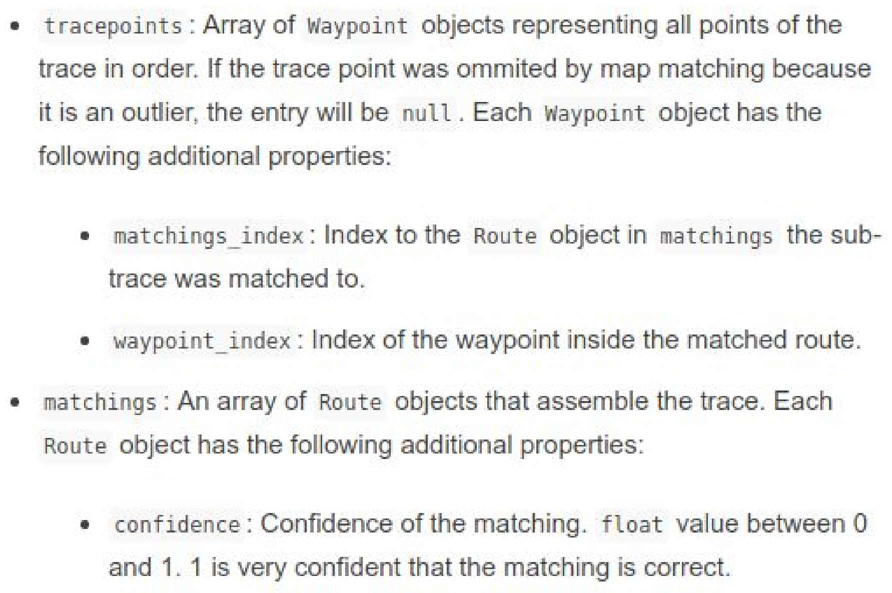
The command above can be used to initialize a web server once the OSRM has been set up. This starts a web server on the port 5000, which can now be used for all queries. Examples of nearest and match queries are mentioned below.

```
curl "http://localhost:5000/nearest/v1/driving/13.388860,52.517037"
```

```
curl "http://localhost:5000/match/v1/driving/13.388860,52.517037;13.385983,52.496891"?steps=True
```

The nearest query inputs only an single coordinate and outputs a waypoints array of Waypoint objects sorted by distance to the input coordinate. Each object has at least the distance property which is Distance in meters to the supplied input coordinate.

The map query inputs a series of coordinates and outputs the following.



- `tracepoints` : Array of `Waypoint` objects representing all points of the trace in order. If the trace point was ommited by map matching because it is an outlier, the entry will be `null` . Each `Waypoint` object has the following additional properties:
  - `matchings_index` : Index to the `Route` object in `matchings` the sub-trace was matched to.
  - `waypoint_index` : Index of the waypoint inside the matched route.
- `matchings` : An array of `Route` objects that assemble the trace. Each `Route` object has the following additional properties:
  - `confidence` : Confidence of the matching. `float` value between 0 and 1. 1 is very confident that the matching is correct.

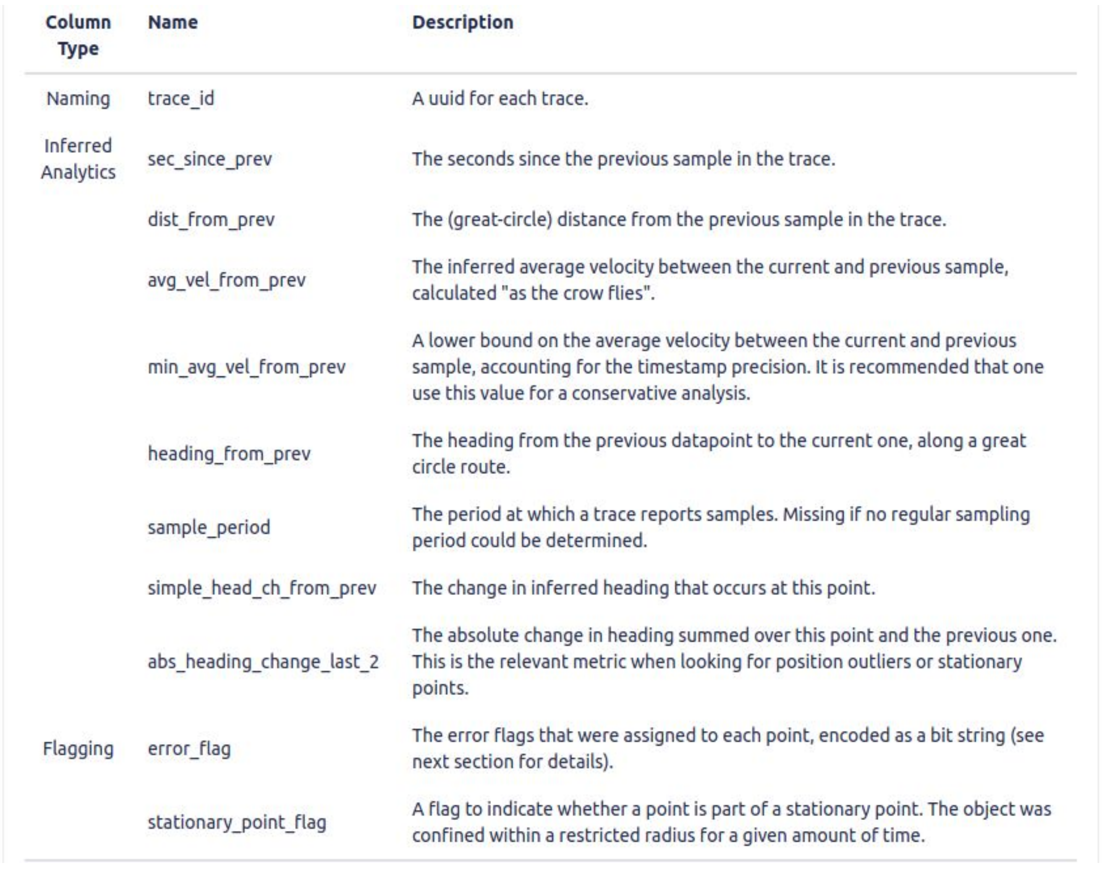
The OSRM wasn’t integrated with the analytical code and that is the next steps for the project. Given that all the subparts are ready and tested, it shouldn’t take a lot of time to integrate them together.

5. Traverse - Error and Stationary point flags

Traverse is a python toolkit for assessing the veracity of trajectory data, to detect and flag simple errors that can be detected based on the properties of each individual trace alone (i.e., without making reference to other traces or to map data). It can be installed and used using the following commands.

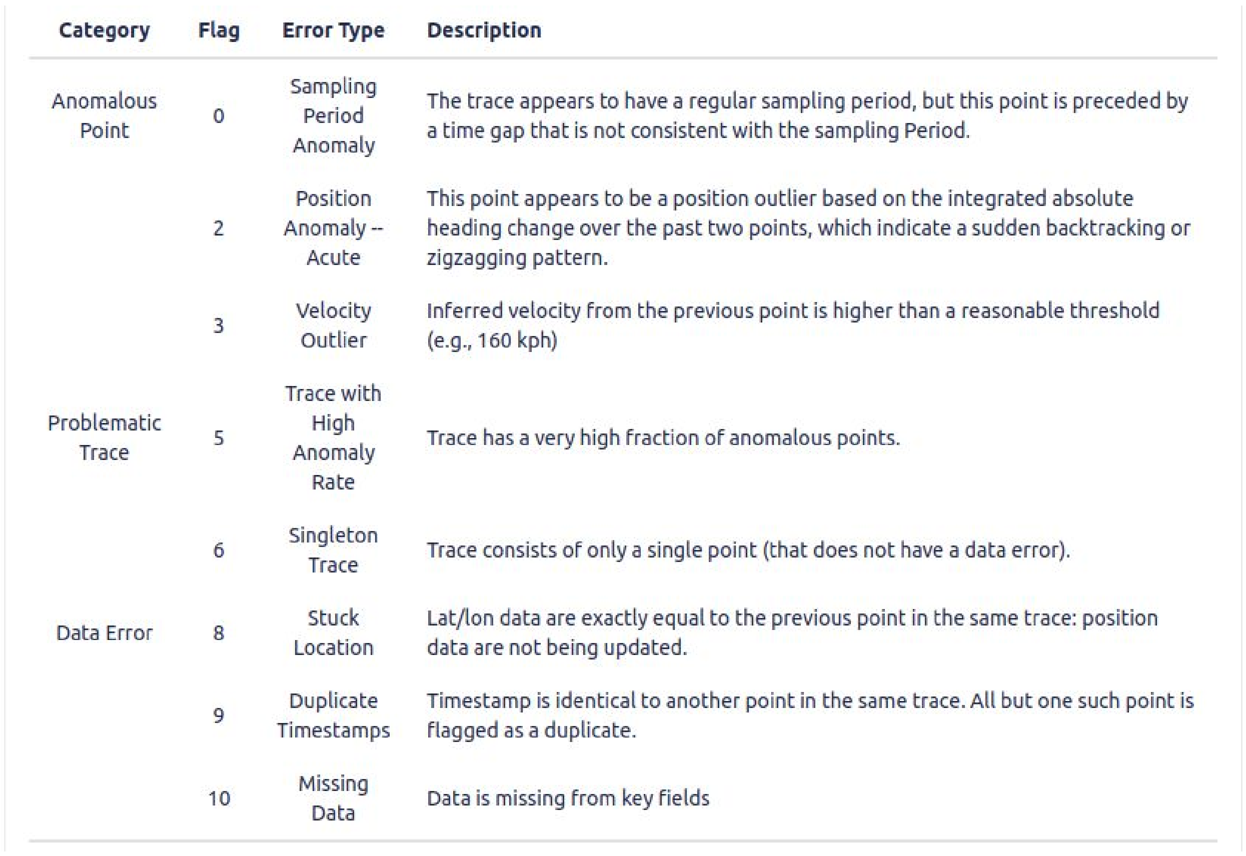
```
git clone https://bitbucket.org/bgerke/traverse.git

pip3 install -e traverse/

python traverse/traverse/pipeline/flag_simple_errors.py p_data.csv
```

The input dataset has rows corresponding to the timestamp, latitude, longitude, and probe ID. The output data set from the pipeline contains a suite of new columns. All positional and velocity columns are inferred using the lat, long, and timestamp columns. The distances and headings are calculated as the crow flies. The description of the output file is mentioned below.



Column Type	Name	Description
Naming	trace_id	A uuid for each trace.
Inferred Analytics	sec_since_prev	The seconds since the previous sample in the trace.
	dist_from_prev	The (great-circle) distance from the previous sample in the trace.
	avg_vel_from_prev	The inferred average velocity between the current and previous sample, calculated "as the crow flies".
	min_avg_vel_from_prev	A lower bound on the average velocity between the current and previous sample, accounting for the timestamp precision. It is recommended that one use this value for a conservative analysis.
	heading_from_prev	The heading from the previous datapoint to the current one, along a great circle route.
	sample_period	The period at which a trace reports samples. Missing if no regular sampling period could be determined.
	simple_head_ch_from_prev	The change in inferred heading that occurs at this point.
	abs_heading_change_last_2	The absolute change in heading summed over this point and the previous one. This is the relevant metric when looking for position outliers or stationary points.
Flagging	error_flag	The error flags that were assigned to each point, encoded as a bit string (see next section for details).
	stationary_point_flag	A flag to indicate whether a point is part of a stationary point. The object was confined within a restricted radius for a given amount of time.

The description of the individual error flag is mentioned below. The library, as can be seen, is a complete pipeline and can be edited to only obtain output as required by editing parts of the code. It has not yet been tested and hence this section only includes how to implement the whole library at this time.



Category	Flag	Error Type	Description
Anomalous Point	0	Sampling Period Anomaly	The trace appears to have a regular sampling period, but this point is preceded by a time gap that is not consistent with the sampling Period.
	2	Position Anomaly -- Acute	This point appears to be a position outlier based on the integrated absolute heading change over the past two points, which indicate a sudden backtracking or zigzagging pattern.
	3	Velocity Outlier	Inferred velocity from the previous point is higher than a reasonable threshold (e.g., 160 kph)
Problematic Trace	5	Trace with High Anomaly Rate	Trace has a very high fraction of anomalous points.
	6	Singleton Trace	Trace consists of only a single point (that does not have a data error).
Data Error	8	Stuck Location	Lat/lon data are exactly equal to the previous point in the same trace: position data are not being updated.
	9	Duplicate Timestamps	Timestamp is identical to another point in the same trace. All but one such point is flagged as a duplicate.
	10	Missing Data	Data is missing from key fields

6. References

1. https://deck.gl/#/documentation/getting-started/installation?section=installation
2. http://kepler.gl
3. https://www.tensorflow.org/api_docs/python/tf/nn/rnn_cell/LSTMCell
4. https://machinelearningmastery.com/time-series-prediction-lstm-recurrent-neural-networks-python-keras/
5. https://www.gakhov.com/articles/find-turning-points-for-a-trajectory-in-python.html
6. https://en.wikipedia.org/wiki/Ramer%E2%80%93Douglas%E2%80%93Peucker_algorithm
7. https://en.wikipedia.org/wiki/Kalman_filter
8. https://dl.acm.org/citation.cfm?id=3151548
9. http://project-osrm.org/docs/v5.5.1/api/#nearest-service
10. https://hub.docker.com/r/osrm/osrm-backend/
11. https://geofabrik.de
12. https://bitbucket.org/bgerke/traverse/src/master/traverse/