Report for Urban Computing project2

Xiang Xinye
SCSE
Nanyang Technological University
Singapore
XI0001YE@e.ntu.edu.sg

Tan Jie Heng Alfred
SCSE
Nanyang Technological University
Singapore
TANJ0307@e.ntu.edu.sg

Yin Wenqi
SCSE
Nanyang Technological University
Singapore
S220157@e.ntu.edu.sg

***Abstract*—In this study we investigate the taxi trajectory patterns within Porto, Portugal, using the city's intricate road network as a foundation. The research consists of six pivotal stages: sourcing the road network and trajectory data, visualizing GPS points, overlaying trajectories onto the road network, enhance the visualization of the route, in-depth route analysis, and refining map matching algorithms. Through these steps, we seek to unravel the complex interplay between taxi movements and urban road layouts. This exploration not only unveils transportation trends but also augments map matching methodologies, paving the way for advanced urban mobility studies in the future.**

I. Introduction

Urban road networks are a intricate web of roads, merging together for urban traffic to flow seamlessly from one destination to the next. These networks tend to be a complex structure, making map matching a challenging task.

Map matching is the process of aligning a sequence of observed user positions to the road network on the map. These positions are typically received as GPS signals, However, because of the complex nature of maps, there could be multiple possible matches for a single observed position. In aggregation, this could lead to multiple possible trajectories for a given sequence of observed positions. In this project, we attempt to preprocess the position data obtained from **??**, before matching these positions to the map using the open-source library *Fast Map Matching*. Finally, with the routes obtained from the map matching, we shall do further analysis to glean some insights into the traffic network within Porto city.

II. Task 1: Data Preparation

In this section, we discuss the data sources utilized in this study and elucidate the process of preparing the data for rigorous analysis.

A. Harnessing OpenStreetMap Data

OpenStreetMap (OSM), a free-to-use geographic database, serves as the primary source of geographic information in this project. In the context of this study, where we focus on Porto City in Portugal, the OSMnx library facilitates extraction and manipulation of data, such as the map data of Porto City. The following code snippet shows the steps to download and visualize the road network of Porto City:

Listing 1: Acquiring and visualizing the map.

```
place = "Porto, Portugal"
# Procure the road network data
start_time = time.time()
G = ox.graph_from_place(place, network_type
    ='drive', which_result=2)
print("Time taken:", time.time() -
    start_time, "seconds")
# Render and archive the road network
ox.plot_graph(G)
ox.save_graphml(G)
```

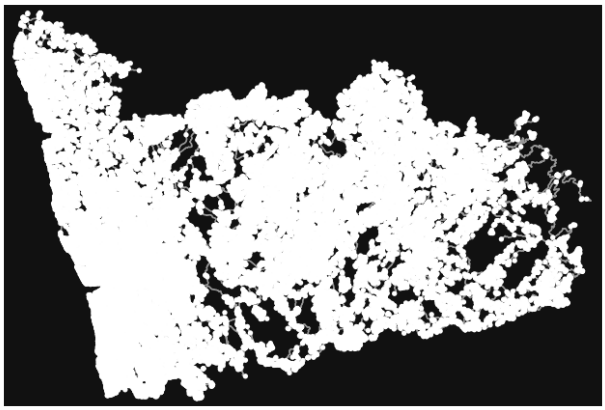


Fig. 1: Visualization of Porto City's road network.

Through OSMnx, the road network data was easily retrieved and stored, allowing us to perform in-depth network analysis. Fig. 1 shows the visualization of the Porto city's road network. The road network has a graphical structure: Each vertex represents a junction or intersection in a road network, while each edge represents a path that connect these intersections. There could therefore be multiple edges between each vertex. From the visualization in 1, we can see that the road network is very dense, with many intersections within the entire Porto City.

B. Taxi Trajectory Data on Kaggle

In this subsection, we elaborate on the taxi trajectory dataset sourced from Kaggle. The tracjetory data used in this research is curated from a Kaggle competition [1], which is only accessible after the creation of a Kaggle account. Within the dataset, there are taxi trajectories, accompanied with other metadata such as whether any GPS location has been dropped (i.e., `missing_data`). For the purposes of this project, we only

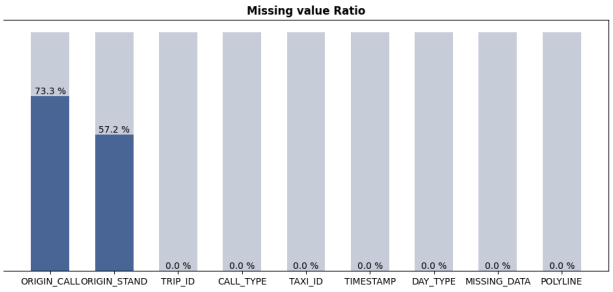


Fig. 2: Analysis missing data.

work with a subset of the first 1500 trips from the train.csv of the dataset. To this end, we extracted the first 1500 taxi trajectories and their associated metadata, placing them into a new CSV file, which we subsequently named as *train-1500.csv*. In the ensuing sections of this study, we will analyze and work on the smaller train-1500 dataset.

III. Task 2: GPS Point Visualization

Before further analysis, it will be helpful to first visualize the taxi trajectories and gain a better understanding of our dataset. In this section, we overlay the GPS points from the taxi trajectory dataset onto the road network of Porto to visualize the some taxi trajectories. We used the OSMnx library, a powerful tool tailored for OpenStreetMap data, to visualize only the first 15 taxi trips, so as to prevent cluttering in the visualization.

A. Data Preparation for Trajectory Boundaries

Prior to visualization, it is essential to preprocess the data. From our dataset, we extract the first 15 trajectories, before computing their encompassing geographical boundaries (delineated by the area these trajectories cover). The subsequent code gives a glimpse into this data processing:

Listing 2: Extraction and boundary calculation for the first 15 trajectories.

```
df_15 = pd.read_csv('data/train-1500.csv',
    nrows=15)
trajectories = []
trajectory_count = 0

for polyline in df_15['POLYLINE']:
    polyline = polyline[2: -2]
    trajectory = []
    if len(polyline) <= 2:
        print(f'Skip empty row')
    else:
        for coordinate in polyline.split('
            ],[']):
            longitude, latitude = map(float
                ,coordinate.strip().split('
                ,'))
            trajectory.append((longitude,
                latitude))
            x_max, y_max, x_min, y_min,
                is_first =
                calculate_boundaries(x_max,
                 y_max, x_min, y_min,
                 is_first, longitude,
                 latitude)
    trajectories.append(trajectory)
    trajectory_count += 1
```

Here, the function `calculate_boundaries` computes the extreme (minimum and maximum longitude and latitude values) values of the set of geographical coordinates. These coordinates allow us to focus on the part of the road network which these 15 trips are taken, to have a better view of how the GPS positions are interspersed on the map.

B. Road Network Acquisition

Upon determining the trajectory boundaries, we then retrieve the section of the road network which contain the 15 trips. To have a better visualization of all the trips, a buffer distance of 0.01 units is applied around the boundaries. We took the following steps for this task:

1) Establishing the boundary coordinates: By extending the longitude and latitude extremes by the buffer distance, we make certain that the extracted road network spans the entire trajectory zone, while have the trajectories visualized not too close to the edge of the image. In the code, we define the extended longitude and latitude as `lower_left_x`, `lower_left_y`, `upper_right_x`, and `upper_right_y`.
2) Constructing a boundary polygon: Using the expanded coordinates, we create a polygon to define our area of interest. This allows us to focus only on the portion of the road network that is relevant to the trips taken.
3) Fetching the road network: Through OSMnx, the 'drive' road network is retrieved, which indicates the vehicular-accessible routes. Furthermore, the road network is restricted to the constructed boundary polygon.
4) Displaying the road network: The procured road network is plotted, giving an outline of Porto's road framework within our specified area.

The accompanying code was implemented from the steps above:

Listing 3: Extraction and depiction of road network.

```
boundary_polygon = Polygon([(lower_left_x,
    lower_left_y), (upper_right_x,
    lower_left_y), (upper_right_x,
    upper_right_y), (lower_left_x,
    upper_right_y)])

G = ox.graph_from_polygon(boundary_polygon,
     network_type='drive')

# Road network plotting
fig, ax = ox.plot_graph(G, show=False,
    close=False, figsize=(12,12))

# Overlay of the first 15 trajectories on
    the road network
color_map = plt.cm.jet(np.linspace(0, 1,
    15))
for i, trajectory in enumerate(trajectories
    [:15]):
```

TABLE I: Sample Taxi Trajectory Data

TRIP ID	CALL TYPE	ORIGIN CALL	ORIGIN STAND	TAXI_ID	TIMESTAMP	DAY TYPE	MISSING DATA	POLYLINE
1372636858620000589	C	NaN	NaN	20000589	1372636858	A	False	[[-8.618643,41.141412],[-8.618499,41.141376],[...
1372637303620000596	B	NaN	7.0	20000596	1372637303	A	False	[[-8.639847,41.159826],[-8.640351,41.159871],[...
1372636951620000320	C	NaN	NaN	20000320	1372636951	A	False	[[-8.612964,41.140359],[-8.613378,41.14035],[-...
1372636854620000520	C	NaN	NaN	20000520	1372636854	A	False	[[-8.574678,41.151951],[-8.574705,41.151942],[...
1372637091620000337	C	NaN	NaN	20000337	1372637091	A	False	[[-8.645994,41.18049],[-8.645949,41.180517],[-...

Note: This is a sample table with a long title that will automatically wrap to the next line when the width is not sufficient.

```
    x_coords, y_coords = zip(*trajectory)
    ax.scatter(x_coords, y_coords, s=30,
        label=f'Trip {i+1}', color=
        color_map[i], edgecolor='k',
        linewidth=0.5)
```

C. GPS Point Integration with Road Network

With the foundational road network in place, supplemented by the GPS points of the 15 trips, we combine them for a unified visualization. Furthermore, in our implementation, we assigned distinct colors to each trip for clear reference. The result is an informative display that highlights the positions of the taxis, contextualized by Porto's road grid (Fig. 3).

In the figure, the white vertices are the vertices for the road network as described above. The other coloured vertices are the GPS locations of the 15 trips (colour coded with the key on the top right of the image). We can see that most of the GPS positions lie on the edges of the road network, which is expected, since the trips are taken on the road (i.e., the edge). However, other GPS readings are noisier and they do not lie directly on any edge. Because of the dense nature of the road network, it is not straightforward when deciding exactly which road the GPS location is referring to. On top of this, some of the GPS locations of the same trip are very close to one another spatially (e.g., trip 14). This appears on the map as just clutter of points and, without more context, it is again not easy to visualize the actual trajectory of the driver. Nonetheless, this rough visualization of the GPS locations allow us to have some insights into the data that we have (i.e., slightly noisy) and how we could further improve the analysis. This paves the way for deeper analysis and map matching in subsequent sections.

IV. Task 3: Map Matching

After gaining some insights into the data we have procured, we can now perform map matching. Map matching is the process of aligning a sequence of observed user positions (i.e., map trajectory data) with the road network on a digital map to produce a sequence of road segments (termed as 'routes') representing the trajectory. This is essential in various applications, such as determining the exact path a vehicle has traveled, even if the GPS data has inaccuracies. For this purpose, the open-source tool Fast Map Matching (FMM) was employed.

A. Background of FMM

In this subsection, we define the problem we hope to solve. Firstly, the road network can be conceptualized as a graph, $G = (V, E)$, with edge set E and vertex set V. The goal of map matching is to find a path, denoted by $route_t = \text{Path}(\hat{p_1}^{(t)}, \hat{p_2}^{(t)}, ..., \hat{p_{N_t}}^{(t)})$, where $\hat{p_i}$ is a projected point of the GPS point p_i, onto the road network, given a fixed and ordered tuple of GPS points, denoted by $trip_t = (p_1^{(t)}, p_2^{(t)}, ..., p_{N_t}^{(t)})$. Here, N_t refers to the number of GPS points in trip t, and the Path$(\cdot)$ is a function that takes into sequences of points to produce an actual trajectory route on the road network.

FMM allows us to efficiently match each sequence of GPS points to a possible route in the road network. This is done by creating an upper bounded origin-destination (hash) table (UBODT) [2]. This table stores information about all pairs of shortest paths, between every two nodes, below an upper bound, Δ. This is done by iteratively applying the single-source Dijkstra algorithm, which is a classical algorithm for finding the shortest path between nodes in a weighted graph. Map matching is therefore done by referencing to the UBODT instead of the more expensive spatial quries.

With the UBODT, we can perform the map matching using a Hidden Markov Model (HMM) based algorithm. In HMM, we assume the following:

- **Hidden states**: These are states, that affect the model, that are not privy to us. In our case, this refers to the road segments.
- **A Markov process**: This is a (strong) assumption that the current state only depends on the previous state.
- **Observations**: These are observable outcomes produced by the model. In this case, this refers to the GPS locations of each trip.

In order to reduce the space of possibilities, we search for at most k candidate points on the road network around each GPS point. In other word, for each $p^{(t)}$, we have $\hat{p}^{(t)} \in C(p, k, r) = \{C_1, C_2, ..., C_s\}$, where $s \leq k$ and C_j is the j^{th} candidate. Here, each $C_j = (p'_j, e, dist, \lambda)$, where e is an edge on the network, p'_j is the projection of point $p_j^{(t)}$ onto the edge, $dist$ is the distance between $p_j^{(t)}$ and p'_j and λ is the distance

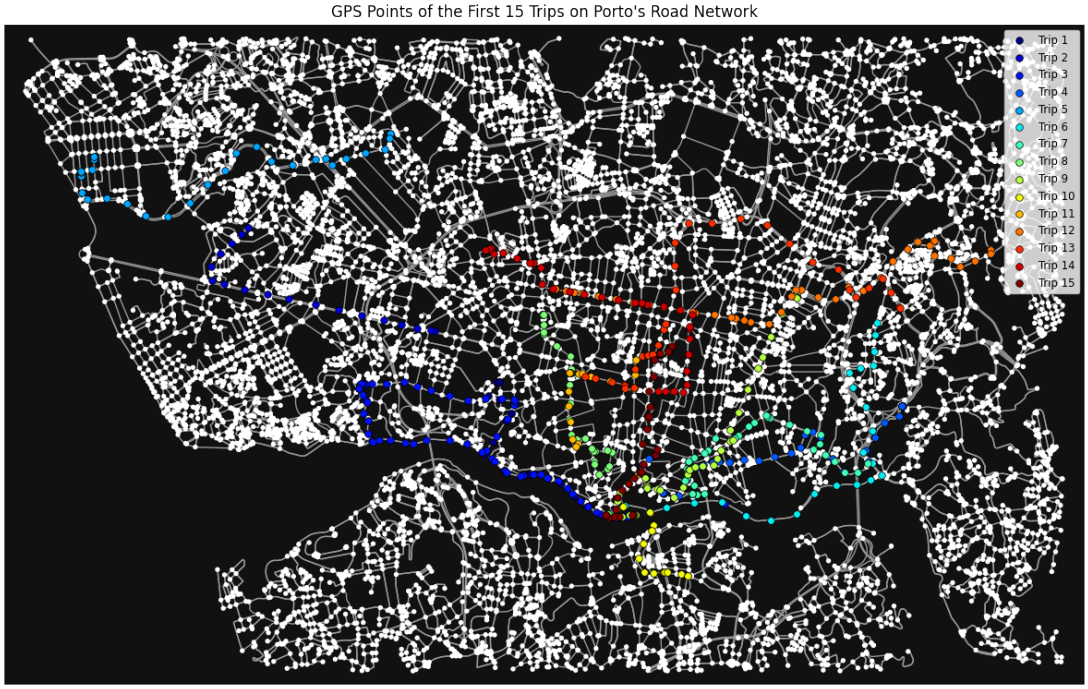


Fig. 3: Visualization of 15 Taxi Trajectories.

of p'_j from the starting vertex of edge e. We also limit the search of these candidates to be at most r units away from p. This means that for each C_j, we have $\|C_j - p\| \leq r$.

With this, we have a list of candidate sets $\{C(p_i^{(t)}, k, r)\}_{i=1}^{N_t}$ for each trip t. This list of candidate sets are therefore the possible sequences hidden states in our HMM, and we are solving for the most probable sequence of hidden states given the GPS points observed.

In order to do this, we have to first find the shortest path (SP) distance between C_i and C_k, for all $C_i \in C(p_{n-1}^{(t)}, k, r)$ and $C_k \in C(p_n^{(t)}, k, r)$. In other words, we have to find all pairwise distance between the candidates of the $(n-1)^{\text{th}}$ GPS point and the candidates of the n^{th} GPS point. These values can easily retrieved from the UBODT. Now, denote $SP_{n-1,n}(i,k)$ as the shortest path distance between C_i and C_k as defined above. Further, we let $d_{n-1,n}$ be the Euclidean distance between p_{n-1} and p_n. Then, following [2], the transition probability is defined as:

$$tp(C_i, C_k) = \frac{\min\ (d_{n-1,n}, SP_{n-1,n}\ )}{\max\ (d_{n-1,n}, SP_{n-1,n}\ )} \tag{1}$$

Generally, transition probability is the likelihood of going from a given hidden state to the next given hidden state. On the other hand, the emission probability is the likelihood of observing a particular observation given the current state. Here, the emission probability, $ep(C_k)$, is given to be

$$ep(C_k) = \frac{1}{\sqrt{2\pi}\sigma} e^{-(C_k.dist)^2/2\sigma^2} \tag{2}$$

where $C_k.dist$ is the $dist$ entry in $C_k \in C(p_n^{(t)}, k, r)$. By modelling this way, we are assuming that the GPS points follows a zero mean Gaussian distribution (i.e., noisy data) and σ is the standard deviation of the GPS error.

With these probabilities and assuming a Markov process, we can calculate the best matched path as,

$$\text{Path}(\hat{p_1}^{(t)}, ..., \hat{p_{N_t}}^{(t)}) = \text{Path}[\underset{i \in \{1,...,N_t\}}{\arg\max} \sum_{n=2}^{N} tp(C_{n-1}, C_n) \times ep(C_n)] \tag{3}$$

where C_{n-1} and C_n are some candidate points in the candidate sets of p_{n-1} and p_n respectively. In other words, we find the sequence $(\hat{p_1}^{(t)}, ...\hat{p_{N_t}}^{(t)})$ by optimizing for both the emission and transition probabilities. This is done by the Viterbi algorithm [3].

Notice, however, that in equation 3 we consider the transition probability between C_{n-1} and C_n. This is afforded by assuming a Markov process, making the computation much less complex. However, in reality, C_n would depend on all C_i for $1 \leq i < n$ (albeit to varying degree), not just C_{n-1}. Specifically, the current point on the road network cannot be entirely independent from all the points that are not the previous point. Therefore, the assumption of Markov process leads to a coarser modelling of reality, as we disregard dependencies further in space and time.

Finally, penalty was implemented to resolve issues of reverse movements (e.g., moving back and forth). Although there are some fringe cases where vehicles do that, such routes are more often than not poorly matched. With this penalty, it enforces that a route is a directed path, instead of a circuit or a cycle.

B. Methodology

With the above background, we use the FMM library in this task.

1) **Data Preparation**: We have to input both the trajectory data and the road network. To ensure compatibility with FMM, we have to indicate the start and end nodes of each edge in the road network. The trajectory data of each trip is made up of a sequence of timestamped GPS coordinates, extracted from the columns of train_1500.
2) **Execution**:
   - FMM was installed and set up according to the documentation provided in its GitHub repository.
   - The trajectory data was input to FMM alongside the road network data.
   - FMM processed the data and mapped each trajectory to the most probable sequence of road segments, forming a route.
3) **Output Verification**: The resulting routes were verified by overlaying them on the road network to ensure they align with actual roads. In the next task, the first 15 routes were visually inspected to further validate the accuracy of the map matching process.

Listing 4: Extraction and boundary calculation for the first 15 trajectories.

```
# Load the network
network = Network(network_path, "fid", "u",
    "v")
print(f"Nodes {network.get_node_count()}
    edges {network.get_edge_count()}")
graph = NetworkGraph(network)

# Generate UBODT if not exists or if
    regeneration is forced
if not os.path.exists(ubodt_path) or
    regenerate_ubodt:
    ubodt_gen = UBODTGenAlgorithm(network,
        graph)
    status = ubodt_gen.generate_ubodt(
        ubodt_path, threshold, binary=False
        , use_omp=True)
    if not status:
        print("Error generating UBODT.")
        return
    print("UBODT generated successfully.")

# Load UBODT
ubodt = UBODT.read_ubodt_csv(ubodt_path)

# Create FMM model
model = FastMapMatch(network, graph, ubodt)
fmm_config = FastMapMatchConfig(k, radius,
    gps_error)
```

C. Results

The map matching process was executed through a tailored Python function leveraging the FMM (Fast Map Matching) library. This function, titled `fmm_map_matching`, serves the purpose of mapping raw GPS trajectories to the nearest road segments in a predefined road network. The function therefore solves for the optimal sequence of hidden states, given a sequence of observations. In order to produce sound and accurate results, the function takes in the following arguments (i.e., parameters):

- **Candidates,** k: This parameter determines the maximum number of road segments candidates to consider for each GPS point. In our implementation, we take k = 6.
- **radius**, r: The value determines the range within which road segment candidates for a given GPS point are searched. In other words, this is the radius of the circle with a GPS point as the centre. The algorithm then performs range query within this circle. Here, we take radius = 0.05.
- **gps_error**: This corresponds to σ seen in equation 2, which is from the assumption that the GPS signals are distributed as a zero-mean Gaussian distribution.
- **threshold**: This is the upper bound of UBODT. With a value of 0.02, it serves to prune the search space during the construction of UBODT.

For the input, the road network data was sourced from the Porto road network, and the trajectory data was retrieved from train-1500. If necessary, the function was also configured to regenerate the UBODT file. Each trajectory from the train-1500 was processed to match it with the road network. The resulting mapped routes, along with details about the matching process, were written to `matched_routines.csv`. In Table II, we show the first five rows in the `matched_routines.csv`.

We exported the following outputs from the `fmm_map_matching` function

1) **Index**: This is a standard index column representing the matched routes.
2) **cpath**: An abbreviation for "complete path", each entry in this column is constructed from opath (see below). In particular, each entry is a list of road segment identifiers that connect the edges in `opath` such that it forms a continuous path. This represents $route_t$ in terms of edges on the road network.
3) **mgeom**: Short for "matched geometry", this is the geometric representation (in LINESTRING format) of the matched paths on the road network. Each LINESTRING is composed of a series of points (longitude, latitude) that represent where the GPS point or trajectory was matched to on the road segment.
4) **opath**: Abbreviation for "optimal path", each entry stores a list of road segment identifiers for the corresponding projected GPS trajectory data (i.e., list of $\hat{p}$ for the trip). This need not form a continuous path.
5) **offset**: This is a list of values representing the relative position of the GPS point, on its corresponding road segment, to the start of the road segment. Therefore, a value close to 0 indicates that the GPS point is near the beginning of its matched segment, whereas a value close to 1 denotes that it is near the end of the segment.
6) **length**: Each entry in this column is a list of values

TABLE II: First five rows of `matched_routines.csv`

Index	cpath	mgeom	opath	offset	length	spdist
0	[1054, 4247, ...]	LINESTRING(-8.6186233 41.141456,...)	[1054, 1054,...]	[0.005765244498603186,...]	[0.007070636992543766,...]	[0.0, 0.0001460268816041218,...]
1	[37924, 179951, ...]	LINESTRING(-8.6398592 41.159752,...)	[37924, 37924,...]	[0.001679683354119456 8,...]	[0.002341297973926323 7,...]	[0.0, 0.0005046527883385151,...]
2	[]	LINESTRING()	[]	[]	[]	[]
3	[]	LINESTRING()	[]	[]	[]	[]
4	[109634, 157753,...]	LINESTRING(-8.6458216 41.180406,...)	[109634, 109634,...]	[0.000449844028692693 97,...]	[0.000586322276888652,...]	[0.0, 4.397033822113258e-05,...]

which represent the length of the corresponding road segment in `cpath`

7) **spdist**: Short for "shortest path distances", this is the $SP_{n-1,n}$ outlined above. For instance, the i^{th} element in an entry, is the distance between $\mathbf{p'}_i$ and $\mathbf{p'}_{i-1}$, where $\mathbf{p'}_j$ is the matched point on the road network for the j^{th} GPS point $\mathbf{p}_j$ in this trip. Note that all the first value of the column is defined to be 0.0.

After map matching, we realized some of the entries are empty. For instance, index 2 in Table II have empty entries for the `cpath`, `mgeom`, and `opath` columns. This is because the has map matching failed to produce a sequence of road segment, given the parameters listed above. For example, if there is just one GPS point within a trajectory which could not be paired with a plausible edge (perhaps r is too restrictive), then the entire trajectory cannot be matched to a route. This could be mitigated by relaxing the constraints, but it might come at the expense of accuracy, as routes produced will be of low confidence (i.e., increased uncertainty threshold). We will further investigate this in the subsequent section.

Nonetheless, our `fmm_map_matching` function, built on the FMM library, successfully and efficiently matched most of the trips to a route on the road network. In order to have a more in-depth understanding, however, we would have to visualize these information on the road network; ultimately, these are temporal data.

V. Task 4: Route Visualization

This section delves into the visualization process of the routes that were generated from the map matching task of the first 15 trips. Visualization is crucial not only for verifying the results but also for a more comprehensive understanding of the spatial relationship between the trajectories and the road network. Here, we aim to produce a clear and accurate representation of the first 15 routes on the road network, distinguishing each trip either by different colors.

A. Methodology

1) **Tool Selection**: The OSMnx tool was primarily chosen for this task due to its compatibility with OpenStreetMap data and its capacity to generate high-quality visualizations. OSMnx facilitates the extraction and visualization of urban street networks from OpenStreetMap, making it suitable for our use-case.
2) **Road segments**: The sequences of road segments obtained from the previous map matching task serve as our primary input. These routes were already mapped to the road network and therefore ready for visualization.
3) **Visualization Process**: We then visualize the first 15 routes in the following manner:
   - Using OSMnx, we extract the road network of the relevant region, similar to what we did in section III.
   - The first 15 routes were then plotted, with different colours, on this road network. The colour-coding allows for a clearer visualization, especially in areas where the routes overlap or intersect.
4) **Validation**: The visualized routes are then compared against the original trajectories and the digital map to ensure they accurately represent the path taken.

B. Results

The visualization can be seen in Fig. 4. Because some of the trips in the first 15 trips of train-1500 were not matched by the function, we also include the subsequent few trips of train-1500. This means that, some of the routes in Fig. 4 do not belong to any of the trajectories shown in Fig. 3.

Looking at the matched trajectories (e.g., the dark blue route in upper right corner of Fig. 4 and the light blue route in upper right corner of Fig. 3), we can see that they are well-matched. From the visualization, we can see that most of the trips went through the city-centre, with only a few going to the fringes of the city. This can be seen from the fact that many of these routes intersect and at times overlap in some segments. These overlapping segments indicate that the road segments are commonly used across many trips. Some routes seem to be exiting the network, however this is because we used the Porto, Portgual road network, and these routes crossed out of the city.

The visualization allows us to have a much better understanding about how the FMM performed, as well as constructing a better picture of how the trajectories were taken. Furthermore, the color-coding distinguishes the different trips, allowing for easier identification of each route. However, it will become infeasible if we were to perform analysis on the visualization, especially if lots of data come into play. For that, we will have to look back at the `matched_routines.csv` file.

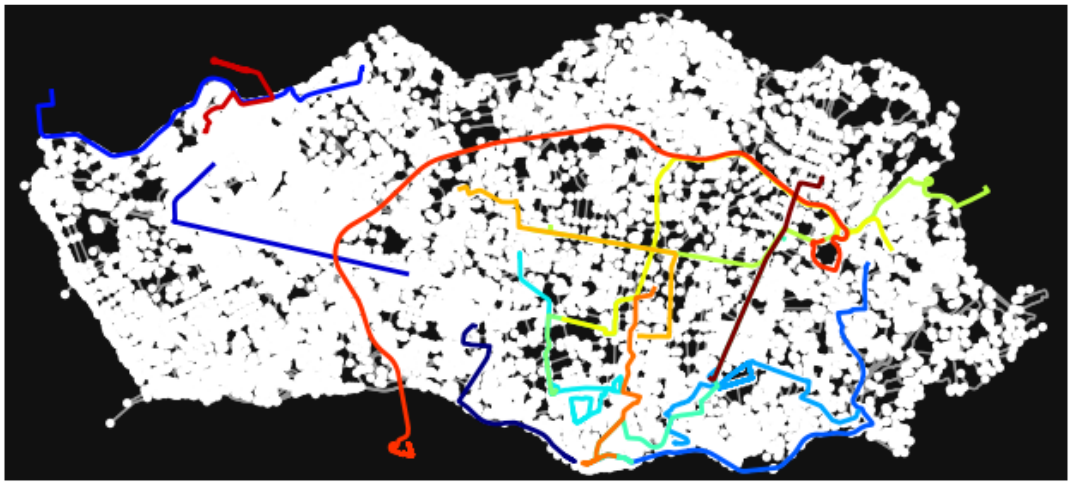


Fig. 4: Visualization of 15 Matched Taxi Routes. The colour coding scheme here differs from Fig. 3.

VI. Task 5: Route Analysis

In this section, we intended to answer two questions:

1) What are 10 road segments that are traversed the most often?
2) What are the 10 road segments that have the longest average travelling time, ignoring those that are not traversed?

A. *Most frequenly traversed road segments*

The idea behind this is simple: We count the number of routes that a road segment contains. This can be done by identifying the road segment within each route. We can identify the road segment by using the entries in `opath` and `cpath` which are identifiers for each road segment. However, each road segment need not have a unique FID – for example, one of the route has $\text{FID}_1 = 1100$ and $\text{FID}_2 = 4231$. If we were to simply count the appearance of all unique FIDs, then the number of traversal for this road will be lower than in reality.

On the other hand, each road segment has a OpenStreetMap ID (OSMID) as well. Unfortunately, it suffers the same issue as that of the FID – some road segments have multiple OSMID. Additionally, some road segments share the same OSMID. In order to properly count the number of traversal per road segment, we have to address these issues. To that end, we produce a new unique identification for each road segment, by mixing in some other information in each road segment.

1) Methodology: Here, we outline how we overcame the issues outlined above. In order to produce a unique ID for each road segment, we perform the following:

1) Given the FID of a road segment, we obtain its OSMID, start node, u, end node, v, and its length, d.
2) We then define a variable `uvd` as a tuple $(\min(u, v)$, $\max(u, v)$, $d)$. This serves as to make the ID unique for each road segment.

With this new, unique ID, we have a one-to-one mapping of each road segment on the road network to an identity. This is a dictionary named `road_fid2id_info`. Now, we can count the number of times a road segment has been traversed as follows:

1) **Create a new dictionary**: We create a dictionary `road_id2frequency`, in order to count the frequency of each unique road segments
2) **Iterate over all the routes**: This counts the frequency of each road segment. This takes the FID from the `cpath` then convert these FIDs to the unique IDs as in `road_fid2id_info`.
3) **Removal of consecutive duplicates**: Since the GPS updates are taken every 15 seconds, there will be routes with repeated road segments in the `cpath`. However, this should only count as one traversal, instead of multiple traversal. Therefore, we have to remove them.
4) **Sorting the frequency of roads**: After iterating over all the routes, we just have to sort the frequency of the road segments in descending order.
5) **Obtain results**: We then return the FID of the top 10 road segments and their counts.

The algorithm to do this is outlined below in VI-A1

Listing 5: 10 most frequently traversed road segments

```
road_id2frequency = defaultdict(int)
for route in results:
    repeat_c_ids = [road_fid2id_info[c][0]
        for c in route['cpath']]
    c_ids = []
    for r in repeat_c_ids:
        if len(c_ids) == 0 or r != c_ids
            [-1]:
            c_ids.append(r)
    for ID in c_ids:
        road_id2frequency[ID] += 1
        # Do the counting
sorted_id = sorted(road_id2frequency.keys()
    , key=lambda x: road_id2frequency[x],
    reverse=True)
# Sort the list by frequency
```

```
result = []
for i in range(5):
    ID = sorted_id[i]
    fids = tuple(road_id2fids[ID])
    road_name = gdf_edges.loc[fids[0], "
        name"]
    result.append((fids, road_name,
        road_id2frequency[ID]))
```

Finally, we have the following ranking. With these, we could visualize the road segments with the highest traversal frequency as in Fig. 7

TABLE III: 10 FIDs road segments that are traversed the most often.

Rank	Road ID	Use Count
1	100535	171
2	4239	164
3	42	147
4	1461	141
5	158673	135
6	1559	134
7	1547	133
8	47	131
9	188463	124
10	50	121

B. Road segments with longest average travelling time

To solve this problem, we have to be able to find the time spent on a road segment. However, we note that all the information given by the FMM algorithm were spatial in nature. Fortunately, we know that the GPS points are updated periodically at every 15 seconds. With some assumptions, we could make some inferences from the information we have.

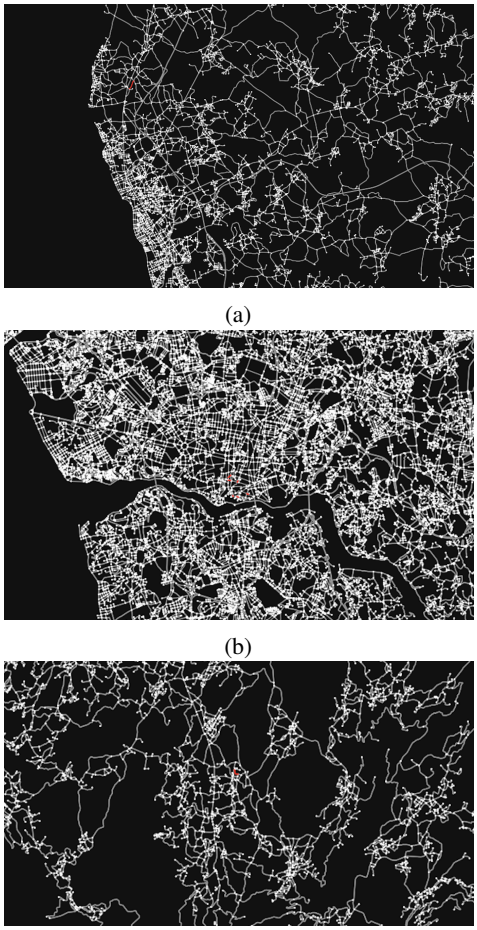
We know that consecutive projected GPS points (i.e., `opatch`) are 15 seconds away. Therefore, we can find the time taken on a road segment by considering pairwise neighbours of the projected GPS point. Note that there are two possible cases: (1) Two consecutive GPS points are matched to the same road segment; (2) Two consecutive GPS points are on different road segments. For the former case, we simply take the time spent on the particular road segment to be 15 seconds.

For for the latter case, where two consecutive projected GPS points are on different road segments, we have to find a way to "distribute" the 15 seconds to all these road segments. In order to do this, we made an assumption that the speed of the vehicle is constant between these two updates. Suppose the road segments that are in between these two projected GPS points are $e_1, ..., e_k$ (these can be found within the entries of `cpath`). Then, we can distribute the travelling time, t_{e_r}, on each road segment as,

$$t_{e_r} = 15 \times \frac{l_{e_r}}{\sum_{i=1}^{k} l_{e_i}}, \tag{4}$$

where l_{e_r} is the length of the road segment that the vehicle covers and can be calculated as,

$$l_{e_r} = \|e_r\| - \text{offset}, \tag{5}$$



(a)

(b)

(a)

Fig. 7: Visualization of the most frequent traversed road segments in red

where $\|e_r\|$ is the length of the road segment and offset is as described in IV.

Suppose that, of all the trips, N_r of them passes through te_r, then there will be N_r of such t_{e_r}. Therefore, we can calculate the average travelling time, T_r, on e_r to be,

$$T_r = \frac{1}{N_r} \sum_{i=1}^{N_r} t_{e_r}^{(i)} \tag{6}$$

1) Methodology: Here, we outline the steps taken to perform what was outlined above:

1) We first create a dictionary called `road_id2travel_patch`. This allows us to store both the time spent on a road segment and the distance between it and its preceding projected GPS point.
2) For each route, we calculate the travelling time for each road segment in this route as outlined above. The travelling time is then updated in `road_id2travel_patch`

In our implementation, we added some constraints in order to clean the data. For instance, we enforced that the `spdist` has to be more than 10^{-7} for every projected GPS point. This is to ensure that the taxis, which could be idling, are travelling at a reasonable speed for it to be considered in our calculation. Furthermore, we removed road segments that have less than 10 routes pass through them, as they are segments that are least travelled, resulting in significantly different time difference. Finally, we also excluded road segments whose total travelled distance by all the taxis is less than 1.5 times the segments' length. Intuitively, this means that the entire road segment is not well covered by all the taxis' routes. For example, the taxis could only be travelling through one specific part of the road segment. Therefore, this time may not be representative.

Listing 6: 10 largest average travelling time road segments

```

    mean_pass_times = dict()  # (log_number, mean_pass_time)
    for ID, logs in road_id2travel_patch.items():
        log_number = len(logs)
        if log_number < ignore_log_number:  # if the number of logs smaller than ignore_log_number, ignore this road ID
            continue
        road_length = road_fid2id_info[road_id2fids[ID][0]][4]
        total_pass_distance = sum([log[1] for log in logs])
        log_rate = total_pass_distance / road_length
        if log_rate < ignore_log_rate:  # if the total distance travelled on this road by cars is smaller than 1.5 length of road, ignore this road ID.
            continue
        total_pass_time = sum([log[0] for log in logs])
        mean_pass_time = total_pass_time / total_pass_distance * road_length
        mean_pass_times[ID] = (log_number, mean_pass_time, log_rate)

    sorted_pass_time = sorted(mean_pass_times.items(), key=lambda x: x[1][1], reverse=True)[:5]
    fid_time_lognumber = [(tuple(road_id2fids[s[0]]), s[1][1], s[1][0], s[1][2]) for s in sorted_pass_time]
```

A snippet of the algorithm can be seen in list:task 5-2. Here, the `log_number` (= 10) is the minimum number of routes that each road segment must have, and `log_rate` (= 1.5) is the ratio of the total distance travelled to road length as mentioned above. Finally, the function returns `fid_time_lognumber` which gives a sequence of road segments' FID with decreasing time travlled on it. We also visualized this result in 8

8 is the visualization.

Listing 7: 10 largest average travelling time road segments

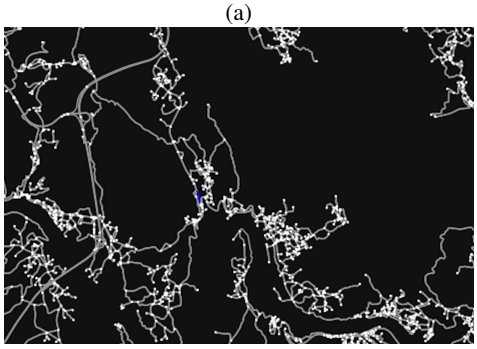
```
road_id2travel_patch = defaultdict(list)

for route in results:
    c_index = 0
    repeat_c_ids = [road_fid2id_info[c][0] for c in route['cpath']]
    c_ids = []
    for r in repeat_c_ids:
        if len(c_ids) == 0 or r != c_ids[-1]:
            c_ids.append(r)
    last_o_id = road_fid2id_info[route['opath'][0]][0]
    for o_index in range(1, len(route['opath'])):
        cur_o_id = road_fid2id_info[route['opath'][o_index]][0]
        if last_o_id == cur_o_id:
# if two consecutive GPS points are in the same road
            if route['spdist'][o_index] > 1e-7:
                road_id2travel_patch[cur_o_id].append((15, route['spdist'][o_index]))
        else:
# if two consecutive GPS passed many different roads
# Omitted: calculate on each of the road respectively before combining

mean_pass_times = dict()
# Omitted: filter out the road segments that is unqualified

sorted_pass_time = sorted(mean_pass_times.items(), key=lambda x: x[1][1], reverse=True)[:10]
# Sort and get the top 10
fid_time_lognumber = [(tuple(road_id2fids[s[0]]), s[1][1], s[1][0], s[1][2]) for s in sorted_pass_time]
```

These two analysis provide useful insight, especially in a dense road network, where some road segments face significant bottleneck issues. For example, after finding out those routes most travelled, in terms of count and time, authorities could use these information to plan future development of the road network to move some vehicles away from these road segments. However, this would require further analysis that is beyond the scope of this project. Nevertheless, these simple analyses provide valuable insights.

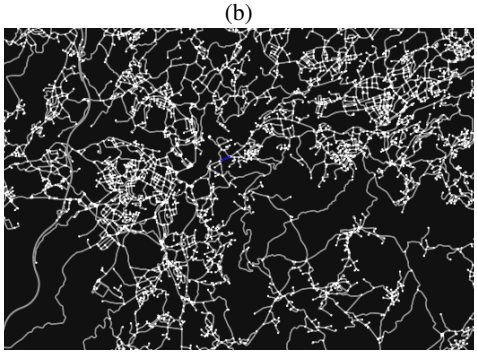
VII. Task 6: Improvements to Map Matching

As discussed in section IV, some of the trajectory data are not being matched, due to the lack of plausible road segments for some of the GPS points in the trip. Put another way, the GPS points are likely noisy data and "too far off" from the actual road network. If we could remove these uninformative data points from the trajectory data, we could potentially match these routes. Furthermore, existing, matched routes, could

(a)



(b)



(c)

Fig. 8: Visualization of road segments with the largest average travelling time in blue

theoretically be less noisy as well. For example, some routes could suffer from some jittering, but by removing the points causing these jitters (some degree of being outliers), the routes could be smoothened out.

A. Outlier detection

We start first by defining what constitutes as an outlier, and have a rough understanding of how many of such outliers there are. There are multiple ways to define an outlier – an example would be to consider the distance of a raw GPS point from an existing road segment. Although feasible, this will be slighlty computationally expensive, since we have to store both the road segments and GPS points in memory, while doing pairwise distance comparison.

We devised a different definition which involves the Euclidean distance between consecutive GPS points. In particular, suppose the i^{th} GPS point, p_i, is an outlier, then $\|p_{i-1} - p_i\| > t$. Intuitively, if it is "too far off" from its previous neighbour, the GPS point is likely to be an outlier, supposing the previous neighbour is not an outlier. In order to decide on this threshold t, our group considered two heuristics:

1) The **distribution of the distances**. This is to ensure that the threshold chosen can both weed out the outliers and keep most of the GPS points.
2) **The frequency of the GPS updates**, which is 15 seconds. This provides a gauge on how plausible certain GPS can be logged.

For the first point, we considered plotting a variogram cloud. However, with the sheer amount of data, this quickly proves infeasible as the visualization will be too dense to be understood. Instead, since we want to understand the distribution, we plotted a histogram (Fig. 9), where the horizontal axis represents the Euclidean distance, in kilometres, between two consecutive poitns, and the vertical axis represents the number of such points.

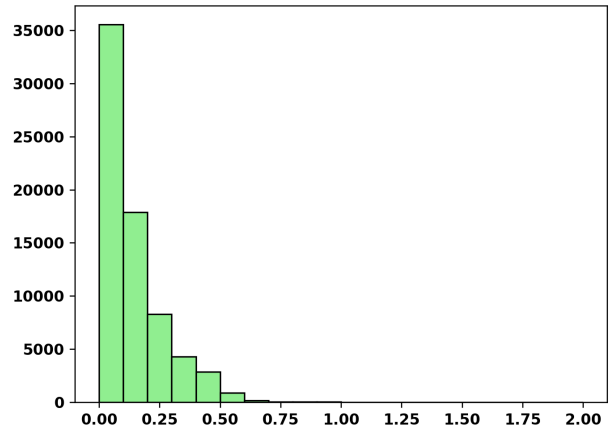


Fig. 9: Distribution of Euclidean distances between two consecutive points. Horizontal axis are in kilometres.

From the histogram, we can see that the Euclidean distance follows a left-tailed distribution – majority of the GPS points is less than 0.5 km from its neighbours. This is realistic, as it translates to speeds less than 120 km/h, with an update frequency of 15 seconds. We could therefore set $t = 0.5$.

B. Methodology

In order to remove the outliers, we first define a $\text{Dist}(\cdot)$ function, which calculates the Euclidean distance between two points, p and p':

$$\text{Dist}(p, p') = 100\sqrt{(p_1 - p'_1)^2 + (p_2 - p'_2)^2} \tag{7}$$

The distance is scaled by 100 as the GPS data were initially at the scale of 10^-2 km. With this, we have the following algorithm:

Listing 8: Outlier removal.

```
1
```

```
#CSV file
train1500 = pd.read_csv('./data/train-1500.csv')

for t in range(1500):
    gps_points = eval(train1500['POLYLINE'][t])
    assert isinstance(train1500['POLYLINE'][t], str)

    #Checking the distances between consecutive GPS points
    if len(gps_points) < 2:
        modified_points = gps_points
    else:
        modified_points = gps_points[:1]
        last_point = gps_points[0]
        threshold = threshold_min
        for i in range(1, len(gps_points)):
            next_point = gps_points[i]
            if distance(last_point, next_point) < threshold: # a valid point
                modified_points.append(next_point)
                last_point = next_point
                threshold = threshold_min
            else:
                threshold += threshold_step
                threshold = min(threshold, threshold_max)
        train1500['POLYLINE'][t] = list2string(modified_points)

#Output trajectories without outlier GPS points as modified_train1500.csv
train1500.to_csv(folder+'/modified_train1500.csv',index=0)
```

Here, instead of using one threshold, t, we have three parameters: `threshold_min`, `threshold_step`, `threshold_max`. By default, each GPS point, p_i, will be checked against its previous neighbour, p_{i-1}: If $\|p_i - p_{i-1}\| <$ `threshold_min` , then we consider p_i as an inlier. Conversely, if $\|p_i - p_{i-1}\| \geq$ `threshold_min`, then we consider p_i as an outlier. When this happens, when it comes to determining if p_{i+1} is an outlier, it will have to be compared with p_{i-1}, since p_i has been removed. But the difference between these two updates is now 30 seconds instead of 15 seconds. Naturally, the threshold should be larger to account for this increased time difference. Therefore, when this happens (i.e., p_i is being removed as an outlier), we add a `threshold_step` to `threshold_min`, in order to 'loosen' the criterion between these two points with larger time difference. Finally, `threshold_max` ensures that the threshold used will always be less than or equal to `threshold_max`.

In our experiment, we used the values `threshold_min` $= 0.5$, `threshold_max` $= 2$, and `threshold_step` $= 0.2$. The choice of `threshold_min` comes from the reasoning that most of the GPS points have less than or equal to that distance, as well as that it is heuristically a reasonable threshold. On the other hand, the `threshold_step` is chosen to be smaller than `threshold_min` and roughly the mean of the Euclidean distance of all the points. This prevents an explosion of the threshold used, while making a reasonable estimate. With these parameters, we obtained a `modified_train1500.csv` file. Subsequently, we generated a `modified_matched_routines.csv` by the map matching algorithm used in section IV. Finally, we generate the visualization of the routes as in the steps used in section V. This can be seen in Fig. 10.

Comparing Fig. 10 to Fig. 3 and Fig. 4, we can see that some trajectories that were previously unmatched (e.g., trip 2), are now able to be matched. By removing these outliers, we managed to provide a candidate road segment to all the the inlier GPS points. This in turns allow us to have a nonzero probability of obtaining the best sequence of road segments for all these routes.

However, looking at the `modified_matched_routines.csv` file, we noticed, again, that some of the trips were once again not matched. For instance, trip 9 does not have any entry in `cpath`, suggesting no routes were assigned to it. This once again means that there exists a GPS point in that trip that is not assigned a road segment. To circumvent this, a naive way would be to lower `threshold_min` once again, to weed out these GPS points. However, a potential problem with this is that some trips may no longer have sufficient GPS points for us to confidently interpolate a route for it. In this case, we suffer some accuracy issues once again, as the sparser the GPS points, the more inaccurate the routes generated will be. A remedy for this would be to remove modified trips with $\leq N$ GPS points entirely.

Regardless, the removal of outliers allow for us to visualize routes that were previously unmatched. However, some routes still remain to be unmatched and other methods of cleaning the data may be required.

VIII. Conclusion

In this project, we worked with visualizing GPS points on the road network of Porto city in Portugal. Furthermore, we used an open-source library called FMM to perform map matching, giving us not only the practical insights of the algorithm, but also the theoretical part of it. Finally, with the matched routes, we were able to perform some detailed analysis on the routes, for example highlighting routes that were most traversed, and improve on the map matching results.

IX. Group Member Contributions

A. Contributions by Xiang Xinye

I have written the task 5 including the code and the report.

B. Contributions by Yin Wenqi

I have written the task 1 and 2 as well as the related code and report.

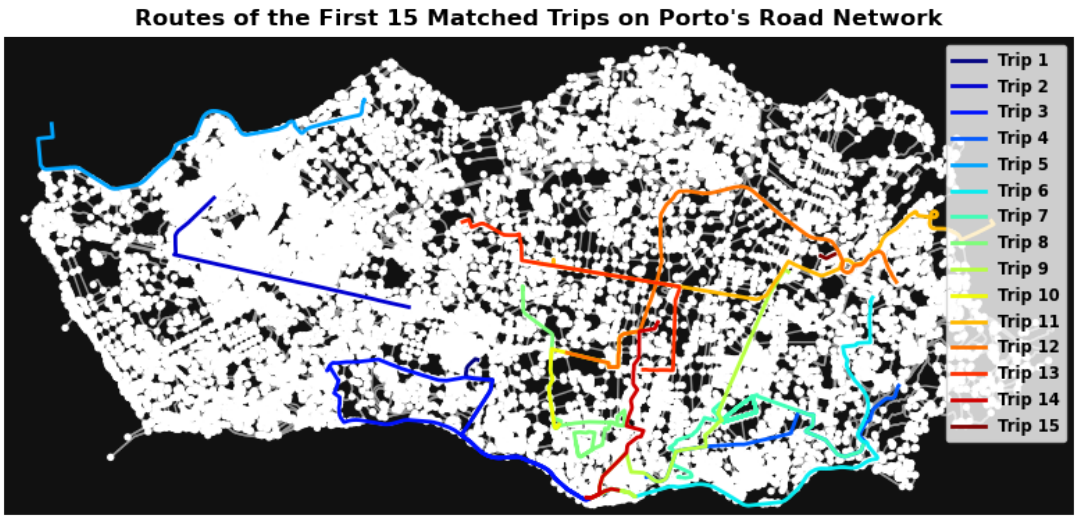


Fig. 10: Visualization of first 15 routes after outlier removal.

C. *Contributions by Tan Jie Heng Alfred*

I have written most of section IV, V, and all of section VII, including the codes and the report. I have also done analysis for the entire report.

APPENDIX

REFERENCES

[1] K. Inc., "Kaggle: Your home for data science," 2023. Accessed on April 20, 2023.

[2] C. Yang and G. Gidófalvi, "Fast map matching, an algorithm integrating hidden markov model with precomputation," *International Journal of Geographical Information Science*, vol. 32, no. 3, pp. 547–570, 2018.

[3] J. G. Forney, "The viterbi algorithm," 1973.