EFFECTS OF TRANSPORT DISASTERS ON HUMAN MOBILITY: DATA, METHODS, FINDINGS

Jacob Delveaux
Submitted 31 July 2020

1 INTRODUCTION

This document is a review of the work done on a PhD project entitled "The Effects of Transport Disasters on Human Mobility" during the period of October 2019 - June 2020. It is broken up in to five sections including the introduction. The second section describes the acquisition of the transport data, including sources for data, types of data used, and how to make TFL API calls. The third section describes the three transport models used during the project, including the assumptions made, data used, and empirical results. The fourth section discusses the data processing and visualization of inputs/outputs. The fifth section is a short discussion on what I have learned during the PhD process. The appendix includes a description of the included python code (see 'code' folder) and the TFL LU/LO (TFL: Transport for London; LU: London Underground; LO: London Overground.) adjacency matrix.

2 Acquisition of Data

Three sources were used to collect data on the London transport network.

1. TFL's Unified API (api.tfl.gov.uk)
2. TFL's NUMBAT Dataset (http://crowding.data.tfl.gov.uk)
3. External information (google, https://www.whatdotheyknow.com/, etc.)
4. TFL Oyster card data

2.1 TFL's Unified API

The TFL Unified API (available at api.tfl.gov.uk) is easy enough to use but not well documented. Requests outside the scope of this document will have to be found either on google or by digging through the TFL blog. I had never worked with an API before the start of this project and so this will be a very basic introduction.

API requests work like a storage service. There are many documents, files, and pieces of data stored on a server (in this case, TFL's servers). You send a letter to TFL describing which piece of data you want and they send the data back to you. Since all of this is done by robots, you have to be very specific directing them to the data you need (e.g. this row, this shelf, this box, and that item.)

It is possible to make API requests through your browser or an application like Postman, although I found it easier to make requests through Python. I expect that this approach will be similar across all languages that support HTML requests and object-oriented programming.

Calls to the API are structured as follows (and found in code/gengraph_lib/tfl_requests.py):

Each request starts with "https://api.tfl.gov.uk/". Then, the user enters a '/' and then can access main directories of the TFL API. Some examples are Line (provides information on the various lines, e.g. Jubilee), StopPoint (provides information on individual stops), AccidentStats (for accident data), etc.

Sub-directories are accessed after the main directory. E.g. Line/Mode/tube will return tube lines, and StopPoint/id will return information on a station with the specified id.

After the directory is accessed, the information can be filtered with '?' followed by specific keywords separated by '&'.

For example: `https://api.tfl.gov.uk/Stoppoint?lat=51.5025&lon=-0.1348&stoptypes=NaptanPublicBusCoachTram&radius=300`

Returns all of the bus stops in a radius of 300 meters from the middle of St. James' Park.

At the end of the call, the user may add '&app_id=0&app_key=1' where 0 and 1 are replaced by their API id and key respectively, which allows for faster requests/data downloads.

The calls are made in python by simply importing the requests library and structuring the request as above. The user may use req = requests.get([request]) to store the request data in a variable, 'req', and req.json() to transform it into a json object.

In Python, json objects are treated like dictionaries with several keys. For example:

```
req = requests.get(https://api.tfl.gov.uk/Line/{0}/Route/Sequence/{1}?serviceTypes=Regular&exclueCrowding=true&app_id=Jubilee&app_key=inbound)

jub_routes = req.json()
```

The jub_routes variable is a dictionary with the keys:

- type
- direction

- isOutboundOnly
- lineId
- lineName
- lineStrings
- mode
- orderedLineRoutes
- stations
- stopPointSequences

Where stations is a dictionary of all StopPoint information for the stations on the Jubilee line, orderedLineRoutes is another dictionary with a dictionary of route information (in this case with Jubilee inbound, there is only Straford -> Stanmore). The nesting structure should be understandable with some practice.

There are a number of calls in the tfl_requests.py that have been used in this project. Namely, getting lines in the network, getting routes on those lines (e.g. Piccadilly has several routes, whereas Jubilee has only one (Stratford - Stanmore (SB) / Stanmore - Stratford (NB)), and getting information on the stoppoints on those routes (gps coordinates, names, naptan ids).

There are two problems, however. The first is that certain stations have multiple names, this is common in the case of stations with Docklands Lightrail. This is helpful if you are considering switches from line to line within stations, although needs to be manually fixed (as it is in gengraph.py) if you are only considering station to station connections. The second problem is that NLCs (National Location Codes) are not included in the API data, for whatever reason. This seems like a strange piece of information for TFL to omit (or, for me to be too foolish to find), considering that their NUMBAT dataset uses the NLCs to identify stations.

2.2 TFL's NUMBAT Dataset

The NUMBAT Dataset provides statistics for passengers within the transport network. A full overview of the included data can be found here: http://crowding.data.tfl.gov.uk/NUMBAT/NUMBAT_Introduction.pdf as of June 2020, information on the journey times and route choice has not yet been released to the public. Essentially, this is a aggregation of data from a week of data collection in August. The files are separated into Monday-Thursday, Friday, and Saturday-Sunday. We have only used Monday-Thursday in this project, although the format is the same.

The NUMBAT data comes in excel files. The two main methods of identifying the stations are the three character ASC code (e.g. Baker Street - BST) and the NLC. Station-to-station information will provide both an ASC and NLC for each station, as well as the direction of travel. The data (number of trains/passengers) is broken into time bands, as described in the document above, and overall is excellently formatted. It can be used directly or converted into a CSV. I have opted to change some of the column names (e.g. Station NLC –> nlc) for easier programming.

2.3 External Information

One piece of information that was not available via NUMBAT or the TFL Unified API was a way of linking NAPTAN (National Public Transport Access Node) codes to NLCs. This information was critical to linking the station-as-a-place information obtained by the API (which uses NAPTAN codes) with the dynamic station information obtained from NUMBAT (which uses NLCs). There had been a file on http://trains.barrycarlyon.co.uk/data/locations/ in 2013, but it has since been removed. I was able to find someone on GitHub who had the file and he was kind enough to email a copy. This file is now present in the data folder and on assenttopeace.com/barrycarltontraindata.

Additionally, if any data sources are missing, the website https://www.whatdotheyknow.com/ allows one to make requests to TFL under the government's freedom of information act. Response times seem fast (< 1 month). I find that these responses are generally lacking and often times users need to request additional data after the initial request.

With external information, there is no standardized format. One may have to go through many processing steps to get the format into a usable form.

2.4 Oyster card data

As part of the TFL API, TFL has released a 5% sample of their oyster card data (i.e. 5% of trips taken on an average week in November 2009) with 2.6 million entries. This data contains the following information:

- day of the week
- transport service (national rail, London underground, etc.)
- starting station
- ending station
- enter and exit time
- fares

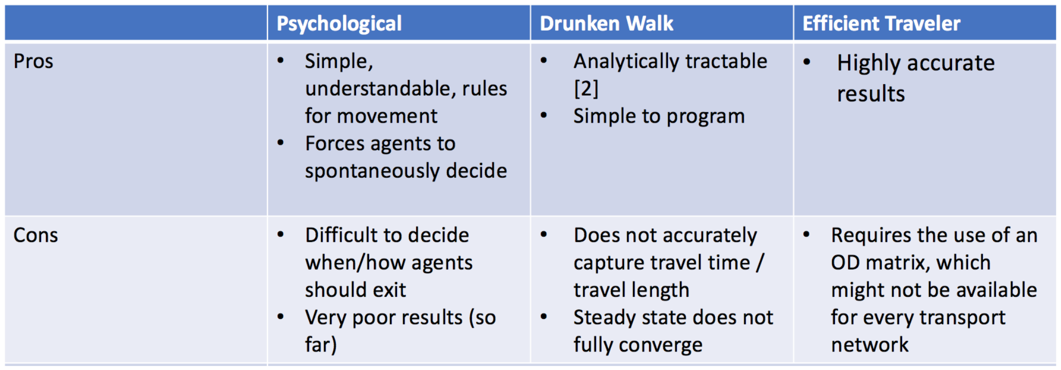
Our first step is to filter the data, removing any entries without start or stop stations and those taken on the bus, leaving 774275 trips. Next, we clean the names on our graph to match those in the oyster card data. This means that we remove things such as commas and apostrophes (e.g. "Queen’s Park" –> "Queens Park"). Afterwards, we can simply iterrate through the entries, collecting the travel time from start and end time and the trip lengths and edge counts from using the Dijkstra algorithm to get the shortest path between the start and end station (an assumption made on the data). Overall, this data set is simple to use.

3 Modelling Human Transport

We have studied three different transport models throughout the project. All of these models were agent-based with variable levels of complexity and required data. A summary of pros and cons is presented in Figure 1. In no particular order, these models are:

1. OD Assignment
2. The Random Walk
3. The Psychological Model

In each section we will state the algorithm and the data required for the model to function. Then, we will discuss the dynamics in detail and highlight the specific choices made. Finally, we present empirical findings from each of the models.



	Psychological	Drunken Walk	Efficient Traveler
Pros	• Simple, understandable, rules for movement • Forces agents to spontaneously decide	• Analytically tractable [2] • Simple to program	• Highly accurate results
Cons	• Difficult to decide when/how agents should exit • Very poor results (so far)	• Does not accurately capture travel time / travel length • Steady state does not fully converge	• Requires the use of an OD matrix, which might not be available for every transport network

Figure 1: Pros and cons of the three models used for human transport

3.1 OD Assignment

Model Algorithm

1. Agents are randomly placed on the network
2. Each is assigned a random destination (described below)
3. Agents move from Origin to Destination via shortest path
4. Repeat for N steps

Data required:

- Station locations and connections
- Origin-Destination travel matrix

Outputs:

- Trip times (km spent travelling)
- Trip lengths (number of stops)
- Edge counts (number of passengers who passed along a given edge)

3.1.1 Assigning the destination

Before running the program, a probability table must be created by running "code/OD_lib/gen_prob_table.py". This takes the TFL OD matrix and creates a matrix where each entry M_{ij} is the probability to transfer to station i given that the agent is at station j. This is done by simply taking the total number of people travelling from j to i and dividing by the sum of all entries.

To get the probability table working, one has to assign numbers to the different nodes in the graph (since python accesses array entries by indexing). This is best done by using $enumerate(tube\_graph.nodes())$ since ordering is preserved.

To select the next destination, the random choice function from numpy is used on the list of ordered NLC codes (corresponding to the enumeration above) using the probability table as a selection weight. The Dijkstra algorithm is used to find the shortest path.

3.1.2 Results

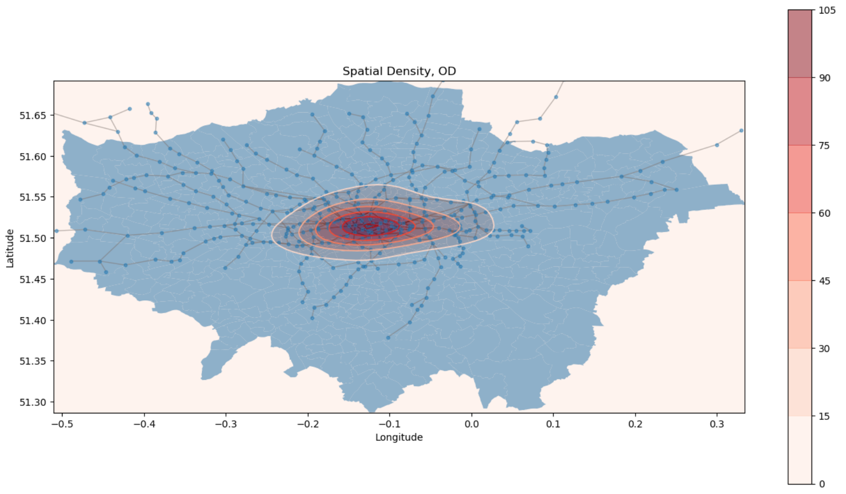


Figure 2: Spatial density of OD model (see 4.2.1)

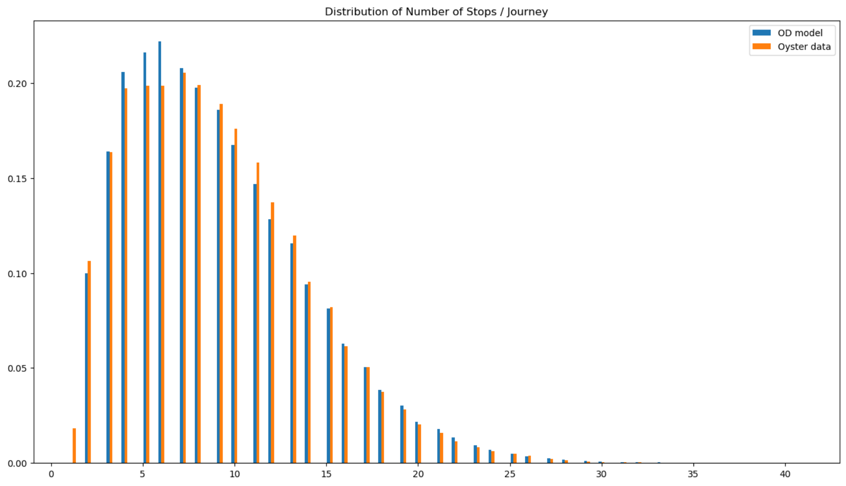


Figure 3: Distribution of number of stops per journey. Blue is the results from the OD model, Orange is TFL Oyster data.

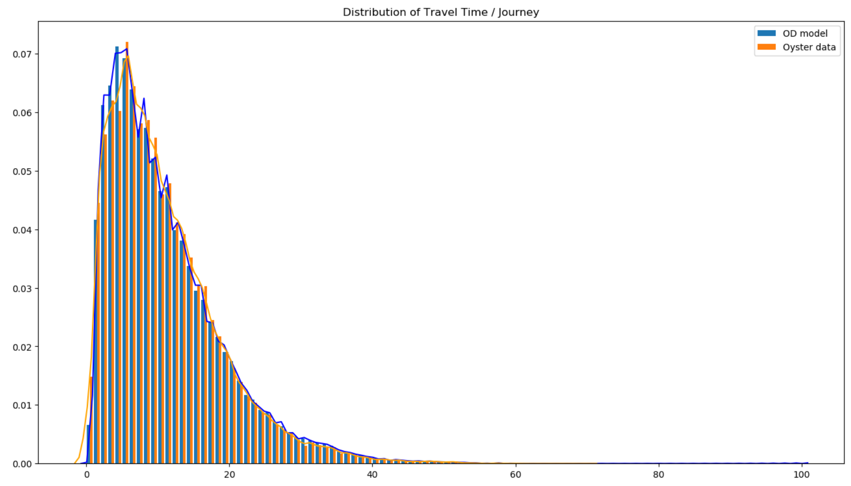


Figure 4: Distribution of travel time per journey. Times are determined by calculating the length of each journey (in km) and dividing by the assumed average speed of 50 km/hr. Blue is the results from the OD model, Orange is TFL Oyster data.

3.2 The Random Walk

Full details of this model can be found in "documents/MEM.pdf"

Model Algorithm

1. One agent is placed randomly on the network
2. At each timestep, the agent randomly moves to another station.
3. This process is repeated N times for M steps

Data required:

- Station locations and connections
- (Optional) Station-to-station flow data

Outputs:

- Edge counts
- Mean first passage times (MFPTs)

3.2.1 Assigning the destination

The next destination is assigned via the numpy random choice function using the list of station indexes from $enumerate(tube\_graph.nodes())$ and weighted by the transition matrix. There are two possible options for the transition matrix: 1) uniform across all connected edges, 2) weighted by the number of passengers travelling across each edge (link-load data). The transition matrix, T, is obtained by $\boldsymbol{T} = \boldsymbol{D}^{-1}\boldsymbol{A}$ where D is the diagonal degree matrix and A is the (weighted or unweighted) adjacency matrix. More details on the dynamics can be found in "documents/MEM.pdf".

3.3 MFPT

There are two ways to calculate the MFPT. First is to use a counting scheme, using the trajectories outputted from the walk and counting how many steps it takes to get from station 1 to station 3 and station 5 to station 97 etc. on average during the first passage. This takes a long time and is implemented using multiprocessing. The alternative solution is to use the eigenvectors and eigenvalues:

$$t_{jk} = \frac{1}{p_j^{eq}} \left[1 + \sum_{l>1} \frac{\lambda_l}{1-\lambda_l} \phi_j^{(l)} (\psi_j^{(l)} - \psi_k^{(l)}) \right]$$

Where t is the matrix of MFPTs, p_{eq} is the equilibrium probability vector ,λ_l is the lth ordered eigenvalue, $\phi_j^{(l)}$ is the jth entry of the lth right eigenvector, and $\psi_j^{(l)}$ is the jth entry of the lth left eigenvector. The eigenvectors and eigenvalues are obtaiend via scipy linear algebra's eig function. The equilibrium probability vector is obtained by $p_{eq} = Tp_{eq}$.

3.3.1 Results

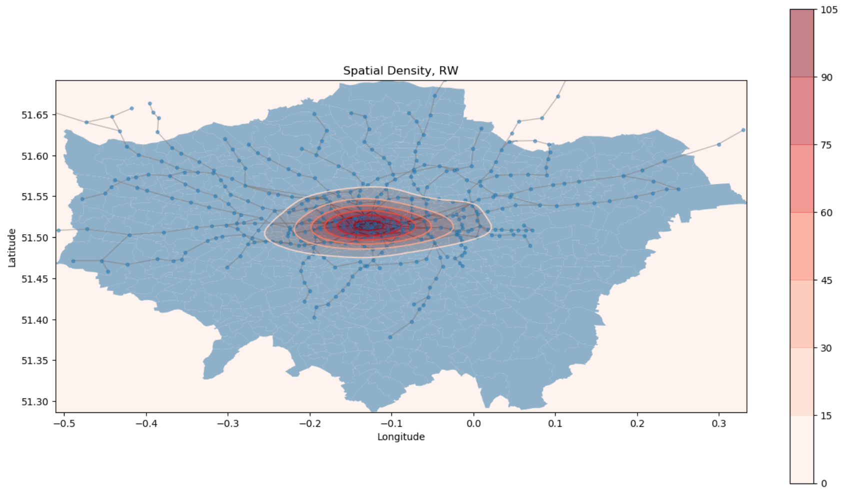


Figure 5: Caption

3.4 The Psychological Model

Full details of this model can be found in "documents/PM.pdf"

Model Algorithm

1. Agents are placed randomly around the network
2. At each station, agents can perform three actions:
   (a) Exit the station (and end the journey)
   (b) Swap trains (e.g. Picadilly -> Central @ Holborn)
   (c) Continue to the next stop
3. When the agent exits, this records their journey and places them randomly on the map Repeat movement for N steps

This model has since been discontinued. It used too much data and had wildly inaccurate results. I do think it is possible to have this model work if the dynamics are streamlined, therefore, I will say the mistakes I ran into and the user may attempt to recreate this model if (s)he wishes.

The first problem was with exiting the stations. Initially, agents would exit the station randomly, with the probability to exit increasing exponentially with their travel time. It was set such that the average travel time was 45 minutes ($Prob = e^{-t/45}$ where e is the exponential function and t is the agent's travel time). This had not worked as well as I had hoped, because it misses out on the real reason people travel, to get to a certain destination. An alternative that was never implemented (but easily could be) was exiting the station based on the number of nearby restaurants/shops/places (more places -> high probability to exit there) or based on TFL entrance/exit data.

The second problem was with the swap probability. The first implementation had a 5% chance to swap at any given station. There is enough data present to obtain realistic swap probabilities on a station to station basis (using the station link flows dataset) which is highly recommended if reproduced.

If these two problems could be fixed, I would consider this a viable model for passenger transport.

3.4.1 Results

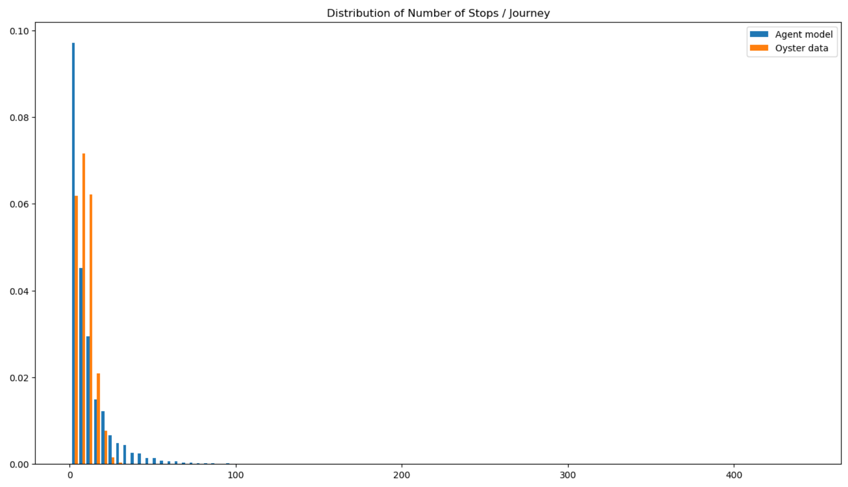


Figure 6: Distribution of number of stops per journey. Blue is the results from the Psychological model, Orange is TFL Oyster data.

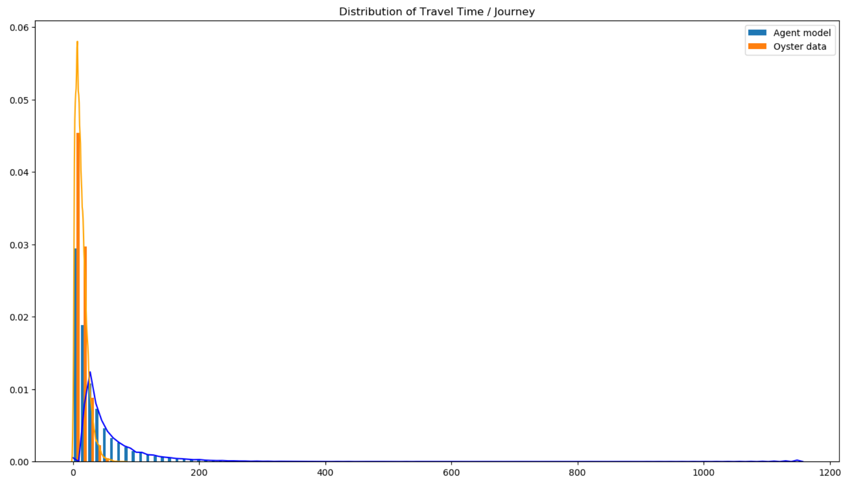


Figure 7: Distribution of travel time per journey. Times are determined by calculating the length of each journey (in km) and dividing by the assumed average speed of 50 km/hr. Blue is the results from the Psychological model, Orange is TFL Oyster data.

4 Data Processing and Visualization

There are two files used for data processing and visualization (outside of the helper files for the various models and creation of the graph described above). These are "print_graph.py" and "TFL_compare.py". We will start with the latter.

4.1 TFL_compare.py

This file processes the outputs from the models described above, namely, travel time, travel length, and edge counts, and compares it with the actual TFL data. The process is simple. First, the program loads in all of the data from the simulation results folders. Then it loads data from the TFL oyster card sample. The data is compared first by making histograms and density plots of the travel time and travel length data. The travel times for simulations are obtained by taking the total distance (in km) and dividing by the average tube speed of 0.83 km/min (50 km/hr). The next comparison is a spatial density plot, as described in the section on "print_graph.py".

4.2 PRINT_GRAPH.PY

This file is used for all of the graph visualizations and spatial density plots. More than anything, this is just a fancy plotting program with easily modifiable parameters, allowing one to plot networkX graphs with color bars, colored nodes, different line sizes, with a background map, etc. etc. The documentation can be found within the code. The spatial density function does warrant a quick explanation however.

4.2.1 SPATIAL DENSITY PLOT

The spatial density plot is based on the edge counts returned by the simulations or extracted from the oyster card data. It works by treating each edge as a mass (located between two stations) with a weight proportional to the number of passengers travelling along that edge. The "midpoint" function gets the location, and the weight is assigned during the "density" function (line 155). The area (min and max bounds of the graph) are divided into a 100x100 grid and fit and a gaussian KDE fits the points. Two contour overlays are added for clairity. I have found that increasing the number of points in the grid does not affect the results in the slightest, at least for the cases studies. One future recommendation would be to shrink the bounding box and observe the spatial density on a smaller scale, which was never attempted. This may provide interesting results and reveal subtle differences between the different simulations.

5 MY THOUGHTS ON THE PHD PROCESS

Unfortunately I had not gained so much from the PhD as I had hoped. Most of the insights below are a result of my other practices in life, readings, etc. that allowed me a easy, relaxed PhD where others seemed constantly and consistently flustered with their progress.

A PhD can be a very rewarding endeavour. If you find that your project is fun and interesting, this is a perfect place to let your creativity shine. Below are some pieces of advice that I found useful in keeping the PhD fun.

The first piece of advice is not to be results oriented. And you won't be. You won't be flustered about the meetings, or wondering what to do, or worried about doing enough if this is something you truly love doing. If you can't be bothered to work on the project each week, or are more concerned about the evaluation of your supervisors than the interesting things you're learning, then consider a different project or career.

The second is that this is your project. I have seen many of my friends fall into the trap of thinking they have to do what their supervisors tell them to do, or that they don't know what their doing because they haven't had their weekly meeting yet. This is not your supervisors project, your supervisors are there to help you with *your* project. It is a great idea to ask your supervisors for assistance, that's what they're there for! They can be a wealth of information and leads. They have spent years trudging through papers and researching topics similar to yours. Let them know if you're having a problem! But don't expect them do do your project for you, and don't think you have to do everything they say. Occasionally, they will suggest things for you to do. Not because they want to torture you or send you off track, but because they want to help.

The third is that there are going to be a lot of ups and downs. Weeks where you get results and have a great time and everything works out and yay! And weeks where you spend hours and hours shuffling through past papers looking for a single equation or manually sorting out a massive heap of data. Both of these are excellent. Don't think of them as good and bad – both are useful. On the slow weeks, take things slow, explore, allow your mind some rest. On the fast weeks, celebrate! On the weeks your happy and working effortlessly, enjoy it! On the weeks your depressed and wondering why you started this project to begin with, use this as motivation to redefine your interest. You chose this project for a reason, what was that reason? Has it changed? What have you learned? Are you unhappy with the project, something in your life, the direction its all going, why?

6 Appendix

6.1 Included resources

There are several folders included with this description file. The first is 'code', which contains all of the python scripts described in this document (see 6.1.1). The second is 'documents' which contains more detailed descriptions of the models, a few meeting notes, and a research proposal for this project. The third is 'tfl_data', which contains some additional data. The third and fourth are 'papers' and 'books' which contain... well... papers and books on related topics.

6.1.1 Code

Inside the code folder there are many files and folders. A1 refers to the psychological model, RW the random walk, and OD the origin-destination. Each model has an additional _lib folder that contains additional functions for the main scripts. The 'data' folder in the parent directory contains all of the data needed for all of the code. GenGraph.py is the script that generates the TFL LU/LO network from API calls. This has an additional _lib folder for additional functions and a folder within 'data'. Oyster_extract.py is the script used to clean and extract oyster card data, as described in section 2.4. Likewise, print_graph.py has been described in section 4.2 helper.py and file_io.py are two additional script files for performing basic functions, reading files, and saving data.

6.2 TFL Adjacency Matrix

A text file of the adjacency matrix can be found in 'code/tlf_adj_mat.txt'

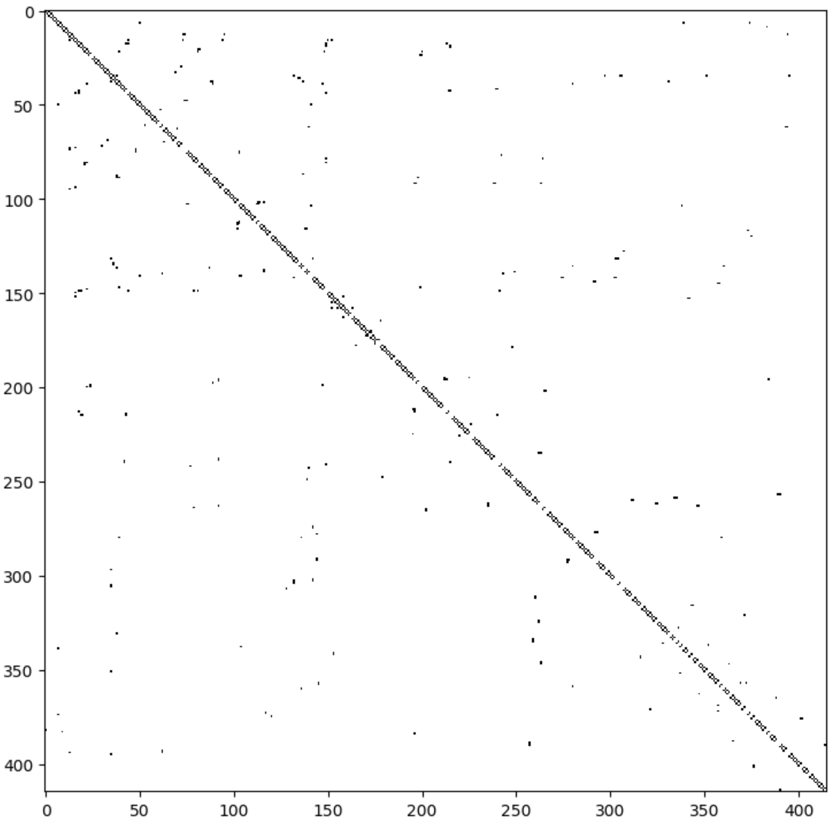


Figure 8: TFL Adjacency matrix