Deep-Learning based model on Travel-Time Estimation

REPORT

Abstract:

With the rise in population, there is an increased demand for vehicles for transportations. This becomes an obvious reason for traffic jams, congestions etc. Therefore, we come forth and present a model that helps us in traffic prediction and gives close to accurate travel time estimation from one location(node) to another. We aim to provide an in-depth comprehension on traffic using deep learning-based tools, thus providing proper solution to the in-hand problem, using the advance technologies to extract the real time data and adopting it to our advantage.

Introduction:

Travel Time Estimation (TTE) module can allow us to perceive the urban traffic situation in advance, and dispatch or control vehicles in real-time based on the feedback information to prevent large-scale congestion.

Traditionally, some classic **time series modelling** models and some data-driven **statistical learning models** were used in this field, especially some ensemble regression tree models.

Although these traditional methods are simple, intuitive and computationally light, they cannot integrate more complex spatial and temporal dynamics, which limits the improvement of accuracy. In recent years, deep learning models have become the fruitful approach for spatial–temporal learning and inference. Although some deep learning models have achieved some breakthroughs, there are still some limitations to be improved. Most of previous deep learning models in TTE are traditional like-

- **Convolutional Neural Networks (CNNs)** and their variants are adopted to extract grid spatial features while
- **Recurrent Neural Networks (RNNs)** and their variants are introduced into capturing temporal dynamics.

We combine these two different scale models for spatial–temporal representation learning, which can be successfully extended to Travel Time Estimation (TTE). However, such methods can have certain drawbacks.

Disadvantages of single deep models or hybrid CNN-RNN based models

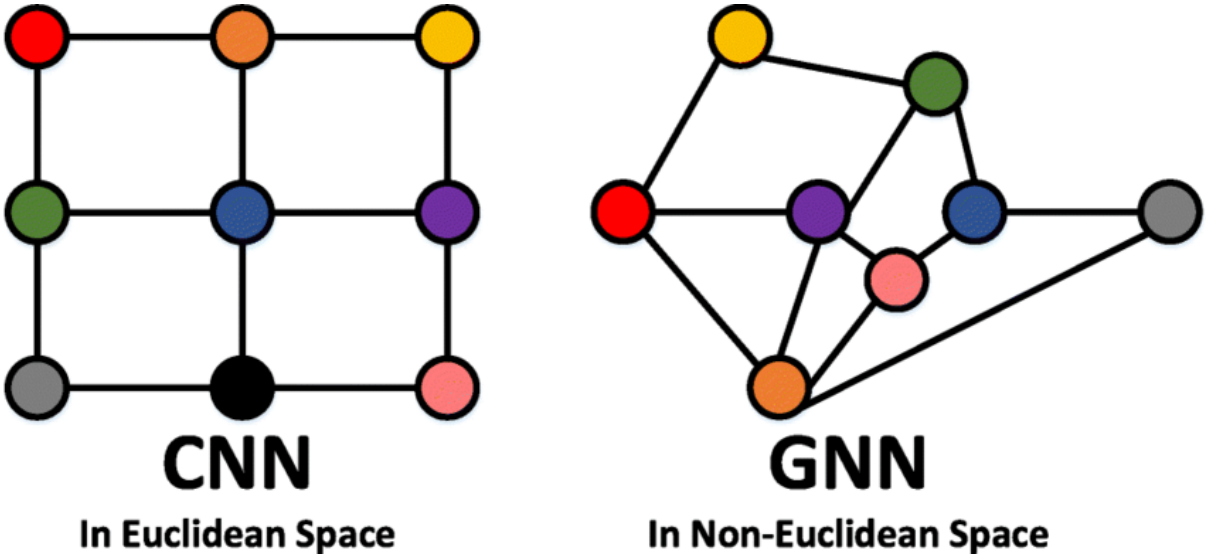
- These models ignore the topology structure of traffic network, which leads to the inability to learn fine-grained spatial–temporal representation.
- Second, most of previous works are difficult to capture multi-scale spatial–temporal dynamics in traffic networks. Many deep learning frameworks capture the high-level

spatial– temporal dynamics by stacking multiple layers, but ignore the impact of shallow-level information at different scales on high-level information.

Advantages of Graph Convolutional Network (GCN)

In the beginning, most of the previous works related to interconnected topological networks were based on the **grid maps** for exploration. The limitation of grid map modelling method can be foreseen when dealing with **non-Euclidean spaces**. Graph Convolutional Networks (GCN), helps us resolve this issue. Graph convolution network **(GCN)** is the promotion of normal convolution network, which can handle the spatial representation learning in non-Euclidean structure.

With the development of **graph deep learning**, Graph Neural Networks (**GNNs**) or Graph Convolutional Networks (**GCNs**) have become the most effective methods for representation learning in spatial topology structure.



Although our entire model revolves around graph deep learning, we have incorporated other methods like **Long Short-Term Memory (LSTM)** and **Autoencoder** for feature vector generation along with **hierarchical clustering** to group and process the data accordingly, the working of which is explained further.

Working:

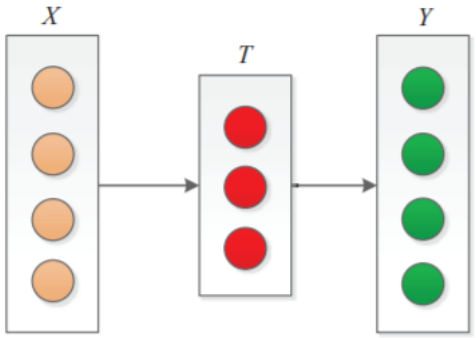
Real-World Data collected and filtered from the taxi trajectories of over 14 thousand taxis in Chengdu (city in China) is used to train and test our proposed model. Statistics and graph attributes of our dataset are as follows-

Latitude range	[30.66, 30.68]
Longitude range	[104.06, 104.08]
Number of trajectories	13442
Number of links	532

Average travel time (s)	246.54
Average moving distance (m)	1417.23
Average Degree	2.5582
Density	0.0048
Clustering coefficient	0.0125

AutoEncoder and Long Short-Term Memory (LSTM)

AutoEncoder is usually used to reduce dimensions or extract features. Given an original input sequence data $X = \{x_1, x_2, \ldots, x_k\}$, where $xi \in R^d$. The characteristic sequence of the original data is obtained through the formula f. T represents the characteristic sequence of the original sequence X, which is defined as $T = \{t_1, t_2, \ldots, t_k\}$, where $t_i \in R^l$. The output of the encoder is used as the input of the decoder. The decoder reconstructs the original data according to the characteristic sequence T. The reconstructed data $Y = \{y_1, y_2, \ldots, y_k\}$, where $y_i \in R^d$. The purpose of decoding is to verify whether the extracted features are valid. After the training of the AutoEncoder is completed, we only use the encoder to extract the characteristics of the original data to obtain more internal structure of the data. The figure shows the basic structure of AutoEncoder.



The encoding and decoding process follows the equations:

$$t_i = f(w_t \cdot x_i + b_t),$$

$$y_i = g(w_y \cdot t_i + b_y)$$

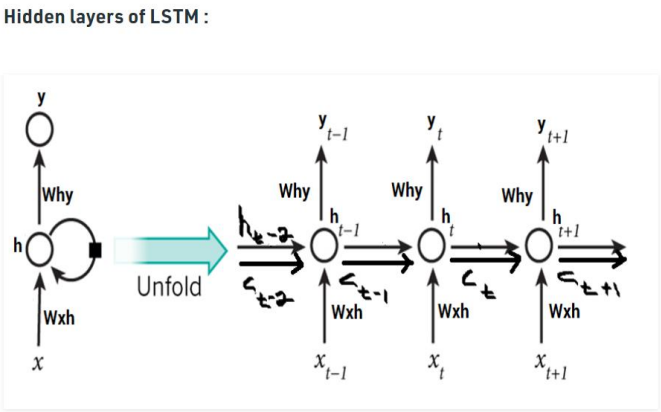
where f (·) and g (·) are the sigmoid functions, and wt , wy and bt , by are weights and biases, respectively. We train the AutoEncoder by minimizing reconstruction error

$$L(X,Y) = \frac{1}{2}\sum_{i=1}^{n} |x_i - y_i|^2$$

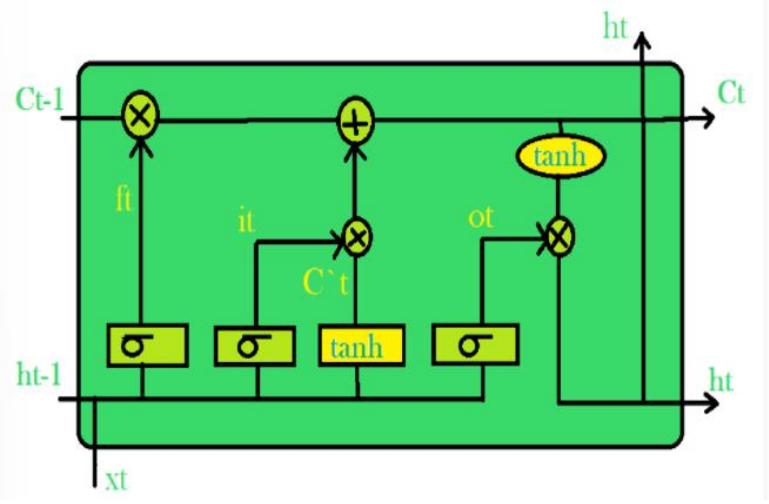
When the difference between the reconstructed data Y and the original data X is small enough, in other words, the output T of the coding process is valid, T is seen as the characteristics extracted from the original data.

Long Short-Term Memory is an advanced version of recurrent neural network (RNN) architecture that was designed to model chronological sequences and their long-range dependencies more precisely than conventional RNNs.

The hidden layer of LSTM is a gated unit or gated cell. It consists of four layers that interact with one another in a way to produce the output of that cell along with the cell state. These two things are then passed onto the next hidden layer. Unlike RNNs which have got the only single neural net layer of tanh, LSTMs comprises of three logistic sigmoid gates and one tanh layer. Gates have been introduced in order to limit the information that is passed through the cell. They determine which part of the information will be needed by the next cell and which part is to be discarded. The output is usually in the range of 0-1 where '0' means 'reject all' and '1' means 'include all'.



The second sigmoid layer is the input gate that decides what new information is to be added to the cell. It takes two inputs h_{t-1} and x_t. The tanh layer creates a vector C_t of the new candidate values. Together, these two layers determine the information to be stored in the cell state. Their point-wise multiplication tells us the amount of information to be added to the cell state. The result is then added with the result of the forget gate multiplied with previous cell state $f_t * C_{t-1}$ to produce the current cell state C_t. Next, the output of the cell is calculated using a sigmoid and a tanh layer. The sigmoid layer decides which part of the cell state will be present in the output whereas tanh layer shifts the output in the range of [-1,1]. The results of the two layers undergo point-wise multiplication to produce the output h_t of the cell.



Benefits of using LSTM

During the training process of a network, the main goal is to minimize loss (in terms of error) observed in the output when training data is sent through it. We calculate the **gradient**, that is, loss with respect to a particular set of weights, adjust the weights accordingly and repeat this process until we get an optimal set of weights for which loss is minimum. This is the concept of backtracking. Sometimes, it so happens that the gradient is almost negligible. It must be noted that the gradient of a layer depends on certain components in the successive layers. If some of these components are small (less than 1), the result obtained, which is the gradient, will be even smaller. This is known as the **scaling effect**. When this gradient is multiplied with the learning rate which is in itself a small value ranging between 0.1-0.001, it results in a smaller value. As a consequence, the alteration in weights is quite small, producing almost the same output as before. Similarly, if the gradients are quite large in value due to the large values of components, the weights get updated to a value beyond the optimal value. This is known as the problem of **exploding gradients**. To avoid this scaling effect, the neural network unit was re-built in such a way that the scaling factor was fixed to one. The cell was then enriched by several gating units and was called **LSTM**.

Use of AutoEncoder and LSTM in Travel Time Estimation

In our model, AutoEncoder is used for feature extraction and LSTM model is used for data prediction.

The AutoEncoder is a kind of unsupervised learning, which can be used as **feature extractor** of our **temporal data**. We determine the initial value of the weight matrix before the training. The weight matrix plays a very important role in the network. we hope to retain the characteristics of the original data after training the weight matrix. If the extracted feature can reconstruct the original data well, it indicates that the features of the original data can be effectively retained through the weight matrix.

Since the upstream and downstream traffic flow will affect the traffic flow at the current location, we need to take these factors into account when **predicting the current traffic flow**. If all upstream and downstream traffic flow data are put into the prediction model, the data dimension will increase and the calculation will be too complicated. To solve this problem, we use the AutoEncoder to extract the characteristics of the upstream and downstream vehicle flow data. In other words, the dimensionality of the traffic flow data is reduced. The acquired characteristics are taken as a part of the input data of the prediction network. In this way, not only the impact of upstream and downstream traffic flow is considered, but also the data dimension is not increased too much.

Implementation

Initial Data

```
wday_off
```

	Link	Node_Start	Longitude_Start	Latitude_Start	Node_End	Longitude_End	Latitude_End	Length	Unnamed: 8	Data1	...	Data41	Data
0	0	0	103.946006	30.750660	16	103.952551	30.756752	921.041014	NaN	89.587892	...	73.957197	88.184
1	1	0	103.946006	30.750660	48	103.956494	30.745080	1179.207157	NaN	121.986947	...	113.506571	113.506
2	2	0	103.946006	30.750660	64	103.941276	30.754493	620.905375	NaN	41.285580	...	59.448387	59.448
3	3	1	104.062539	30.739077	311	104.060024	30.742693	467.552294	NaN	37.796143	...	38.472874	46.950
4	4	1	104.062539	30.739077	1288	104.062071	30.732501	730.287581	NaN	60.548947	...	80.810511	80.810
...	...	...	...	...	...	...	...	...	...	...	...	...	
5938	5938	1900	104.029570	30.670992	1729	104.031714	30.670676	208.333343	NaN	15.136227	...	18.137848	21.428
5939	5939	1901	104.025686	30.626472	1233	104.023241	30.622808	468.731933	NaN	40.644760	...	47.036515	43.083
5940	5940	1901	104.025686	30.626472	1565	104.023113	30.627839	289.473210	NaN	50.246073	...	37.986764	39.523
5941	5941	1901	104.025686	30.626472	1658	104.028963	30.624882	360.123488	NaN	36.935304	...	49.169831	40.066
5942	5942	1901	104.025686	30.626472	1833	104.026919	30.628522	256.113864	NaN	26.720084	...	33.190697	32.782

Temporal data extracted from initial data

```
]: wday_off_data
```

]:

	Data1	Data2	Data3	Data4	Data5	Data6	Data7	Data8	Data9	Data10	...	Data41	Data42	
0	89.587892	89.587911	89.587872	89.587949	89.587796	89.588103	89.587489	89.588717	89.586260	89.591175	...	73.957197	88.184778	72
1	121.986947	121.986947	121.986947	121.986947	121.986947	121.986947	121.986948	121.986947	121.986948	121.986945	...	113.506571	113.506571	140
2	41.285580	41.304056	41.267120	41.341058	41.193446	41.489733	40.901361	42.095280	39.773298	44.705187	...	59.448387	59.448387	59
3	37.796143	37.796143	37.796143	37.796143	37.796143	37.796143	37.796143	37.796143	37.796143	37.796143	...	38.472874	46.950858	50
4	60.548947	60.548947	60.548947	60.548947	60.548947	60.548947	60.548947	60.548947	60.548947	60.548947	...	80.810511	80.810511	76
...	...	...	...	...	...	...	...	...	...	...	...	...	...	
5938	15.136227	21.047709	40.687163	18.374817	18.598035	30.906595	22.455091	16.746485	23.017850	23.148149	...	18.137848	21.428572	21
5939	40.644760	39.590819	40.393416	40.369257	40.417604	87.431863	40.722412	43.042589	42.322098	51.953344	...	47.036515	43.083732	48
5940	50.246073	50.246073	50.246073	50.246073	50.246073	50.246073	50.246073	22.638021	50.246073	50.246073	...	37.986764	39.523523	39
5941	36.935304	33.306219	37.656066	43.470737	32.411114	33.199605	37.907736	37.572658	41.753448	62.509381	...	49.169831	40.066719	40
5942	26.720084	26.715246	26.981895	34.148515	26.249392	32.200579	26.324337	27.834261	31.512728	30.654473	...	33.190697	32.782575	33

Autoencoder model to extract most important features from temporal data. Data reduced from 50 features to 16

```
[22]: model = Sequential([
        tf.keras.layers.Dense(48, activation='relu', input_shape=(1,n_features)),
        tf.keras.layers.Dense(32, activation='relu'),
        tf.keras.layers.Dense(16,activation='relu'),
        tf.keras.layers.Dense(32, activation='relu'),
        tf.keras.layers.Dense(48, activation='relu'),
        tf.keras.layers.Dense(n_features)
      ])
      model.compile(optimizer='adam', loss='mse')
```

Generated feature vectors

In [51]:
```
feature_vectors = pd.DataFrame(bottleneck_activation,columns=wday_off_data.columns[:16])
```

In [52]:
```
feature_vectors
```

Out[52]:

	Data1	Data2	Data3	Data4	Data5	Data6	Data7	Data8	Data9	Data10	Data11	Data12	Data13	Data
0	116.082909	123.622398	85.287445	149.549911	174.133942	188.084885	9.545237	147.277481	0.0	0.0	0.0	0.0	79.887703	86.4161
1	159.559021	132.381653	84.351204	219.202988	197.407440	333.639679	120.090324	218.795212	0.0	0.0	0.0	0.0	68.866356	145.7944
2	79.968361	85.382072	82.910690	99.203796	116.550484	119.644218	0.000000	77.781624	0.0	0.0	0.0	0.0	23.115873	20.5838
3	44.305367	35.493378	56.044086	87.022240	87.019447	76.934547	12.407438	49.449409	0.0	0.0	0.0	0.0	48.313087	40.2031
4	119.960312	113.504059	111.679352	132.575348	148.718277	186.764053	10.403971	104.110809	0.0	0.0	0.0	0.0	37.981537	47.9681
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
5938	30.393410	23.199553	15.879178	42.399460	38.844337	48.568913	6.120677	39.423615	0.0	0.0	0.0	0.0	20.139940	21.4630
5939	54.055328	71.285126	52.779961	88.048149	107.290245	106.328407	9.251759	72.109489	0.0	0.0	0.0	0.0	49.936356	41.9039
5940	40.333332	70.569580	23.340073	72.328934	109.610283	90.649696	14.924143	76.440002	0.0	0.0	0.0	0.0	28.096336	54.8979
5941	59.759495	64.389412	47.380970	89.560692	104.454933	89.645668	3.077206	56.068260	0.0	0.0	0.0	0.0	34.335976	28.7013
5942	41.041813	61.611794	1.681240	57.905613	43.561924	107.774307	50.766102	59.250950	0.0	0.0	0.0	0.0	0.000000	28.8147

5943 rows × 16 columns

Extraction of spatial data

In [58]:
```
wday_off_spatial
```

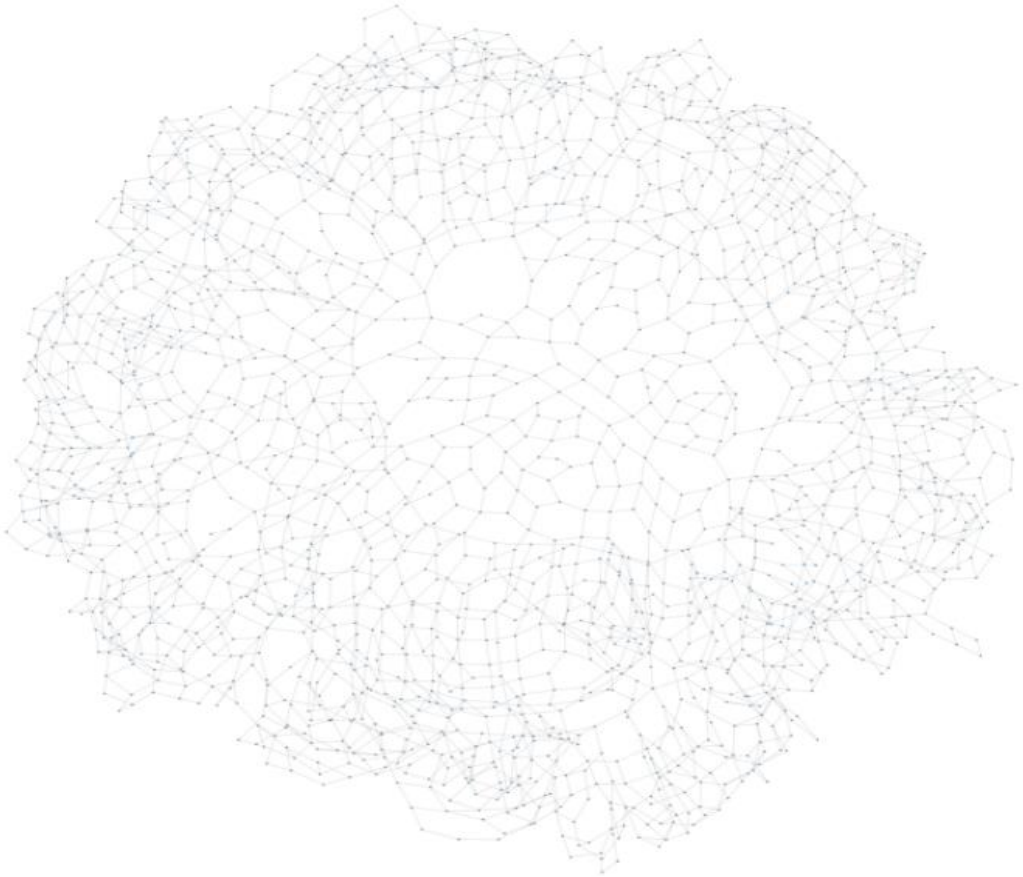
Out[58]:

	Link	Node_Start	Longitude_Start	Latitude_Start	Node_End	Longitude_End	Latitude_End	Length	Time
0	0	0	103.946006	30.750660	16	103.952551	30.756752	921.041014	85.424151
1	1	0	103.946006	30.750660	48	103.956494	30.745080	1179.207157	125.386769
2	2	0	103.946006	30.750660	64	103.941276	30.754493	620.905375	53.618874
3	3	1	104.062539	30.739077	311	104.060024	30.742693	467.552294	40.787167
4	4	1	104.062539	30.739077	1288	104.062071	30.732501	730.287581	77.209144
...	...	...	...	...	...	...	...	...	...
5938	5938	1900	104.029570	30.670992	1729	104.031714	30.670676	208.333343	21.247776
5939	5939	1901	104.025686	30.626472	1233	104.023241	30.622808	468.731933	47.172532
5940	5940	1901	104.025686	30.626472	1565	104.023113	30.627839	289.473210	41.050324
5941	5941	1901	104.025686	30.626472	1658	104.028963	30.624882	360.123488	42.633222
5942	5942	1901	104.025686	30.626472	1833	104.026919	30.628522	256.113864	31.851535

5943 rows × 9 columns

Generating a graph for our spatial data

```
In [102]:
import networkx as nx
# Since the actual graph is undirected, we convert it for visualization purpose.
nx_g = g.to_networkx().to_undirected()
plt.figure(figsize=[150,150])
pos = nx.kamada_kawai_layout(nx_g)
nx.draw(nx_g,pos, with_labels=True)
```



Clustering

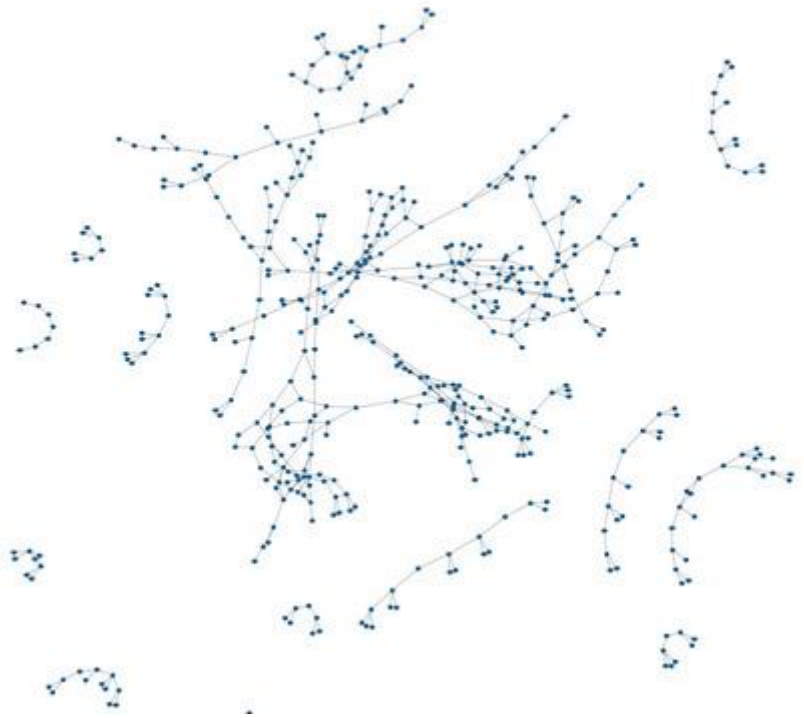
We will be implementing clustering. By clustering we mean that we will be grouping the data accordingly so that we could be able to group a particular set of nodes which could be then easily takes those nodes in as a group, therefore carrying out the task, such as extracting a particular set of nodes will take less time. There are many methods of clustering. The one we will be using is **Hierarchical clustering**.

Hierarchical clustering (also called hierarchical cluster analysis or HCA) is a method of cluster analysis which seeks to build a hierarchy of clusters. Specifically, we will approach using, divisive method or “top-bottom” method: all observations start in one cluster, and splits are performed recursively as one moves down the hierarchy.

The mostly used algorithm has a time complexity of $O(n^3)$ and requires $\Omega(n^2)$ memory. This shows that the algorithm is too slow even if we take a medium scale dataset into consideration. However, we can possibly tweak the algorithm so that we can reduce the time complexity of the clustering to $O(n^2.\log n)$, This time could be taken down even lower by using faster heuristics to choose splits, such as k-means, etc. Still, the dataset that we will be using, the time complexity can be said as quite big. But considering the fact that topology of

the area would not change drastically, we can run clustering once and then we consider the clusters as object and could be used for preparing the model. Although, if new routes are added in those areas, we might need to re-run the clustering segment.

Clustered Data



GNN