Assessment for CASA0002 – Urban Simulation

https://github.com/ruicixia1/simulation

Part 1: London’s underground resilience

I. Topological network

I.1. Centrality measures:

In [1]:
```
# we will import all the necessary libraries
import pandas as pd
import numpy as np
import geopandas as gpd
import networkx as nx
import matplotlib.pyplot as plt
from matplotlib.pyplot import cm
import json
import re
from shapely.geometry import Point, LineString #this library is for manipulating geometr
from scipy.spatial import distance
```

```
C:\Users\lenovo\AppData\Local\Temp\ipykernel_38124\2494671397.py:2: DeprecationWarning:
Pyarrow will become a required dependency of pandas in the next major release of pandas
(pandas 3.0),
(to allow more performant data types, such as the Arrow string type, and better interope
rability with other libraries)
but was not found to be installed on your system.
If this would cause problems for you,
please provide us feedback at https://github.com/pandas-dev/pandas/issues/54466

  import pandas as pd
```

Degree centrality, betweenness centrality and eigenvector centrality are the three most useful measures for identifying the most important nodes in the underground networks.

1. **Degree Centrality**: Measures the number of direct connections a node has and is useful for indicating the local importance of a node (Bloch, Jackson, and Tebaldi, 2019). Stations with high degree centrality are crucial as they serve as major hubs, connecting various lines and facilitating a high number of passenger movement volumns across the network. A station with high degree centrality can significantly impact the network's functionality if removed or disrupted. ($deg(v)$ is the degree of vertex v, and N is the total number of vertices in the graph)

$$C_D(v) = \frac{deg(v)}{N-1}$$

1. **Betweenness Centrality**: Quantifies the number of times a node acts as a bridge along the shortest path between two other nodes (Freeman, 1977). In the context of the underground, stations with high betweenness centrality are critical for maintaining network efficiency. They often serve as essential transfer points, and their removal could isolate network segments and increase travel times

dramatically. (σst is the total number of shortest paths from node s to node t and $\sigma st(v)$ is the number of those paths that pass through v .)

$$C_B(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

Normalised:

$$C_B(v) = \frac{2}{(n-1)(n-2)} C_B(v)$$

1. **Eigenvector Centrality**: This measure not only considers the number of connections a station has but also the quality of those connections (Bonacich, 1987). A station connected to other highly connected stations has higher eigenvector centrality. This is crucial for identifying influential stations within the network, highlighting nodes that, while they may not have the highest number of direct connections, are strategically positioned.

$$C_E(v) = \frac{1}{\lambda} \sum_{t \in M(v)} C_E(t)$$

$$\mathbf{A}\mathbf{x} = \lambda \mathbf{x}$$

In [2]:
```
G = nx.read_graphml('london_updated.graphml')
```

In [3]:
```
for node, data in G.nodes(data=True):
    # convert the string to a tuple
    coords_str = data['coords'].strip("()")
    # add the new key to the dictionary
    data['coords'] = tuple(map(float, coords_str.split(',')))
```

In [4]:
```
# check
is_connected = nx.is_connected(G)
print(f"Graph is connected : {is_connected}")
```

```
Graph is connected : True
```

In [5]:
```
# To check node attributes:
#G.nodes(data=True)
```

In [6]:
```
# To check edge attributes:
#G.edges(data=True)
```

In [7]:
```
print(G.number_of_nodes())
print(G.number_of_edges())
```

```
401
467
```

In [8]:
```
# Let's plot the tube network!

# We can plot the tube network with the names of the stations as labels
fig, ax = plt.subplots(figsize=(25,20))

node_labels = nx.get_node_attributes(G, 'station_name')

pos = nx.get_node_attributes(G, 'coords')

nx.draw_networkx_nodes(G,pos,node_size=50,node_color='b')
nx.draw_networkx_edges(G,pos,arrows=False,width=0.2)
```

```
nx.draw_networkx_labels(G,pos, node_labels, font_size=10, font_color='black')

plt.title("London tube network",fontsize=40)
plt.axis("off")
plt.show()
```

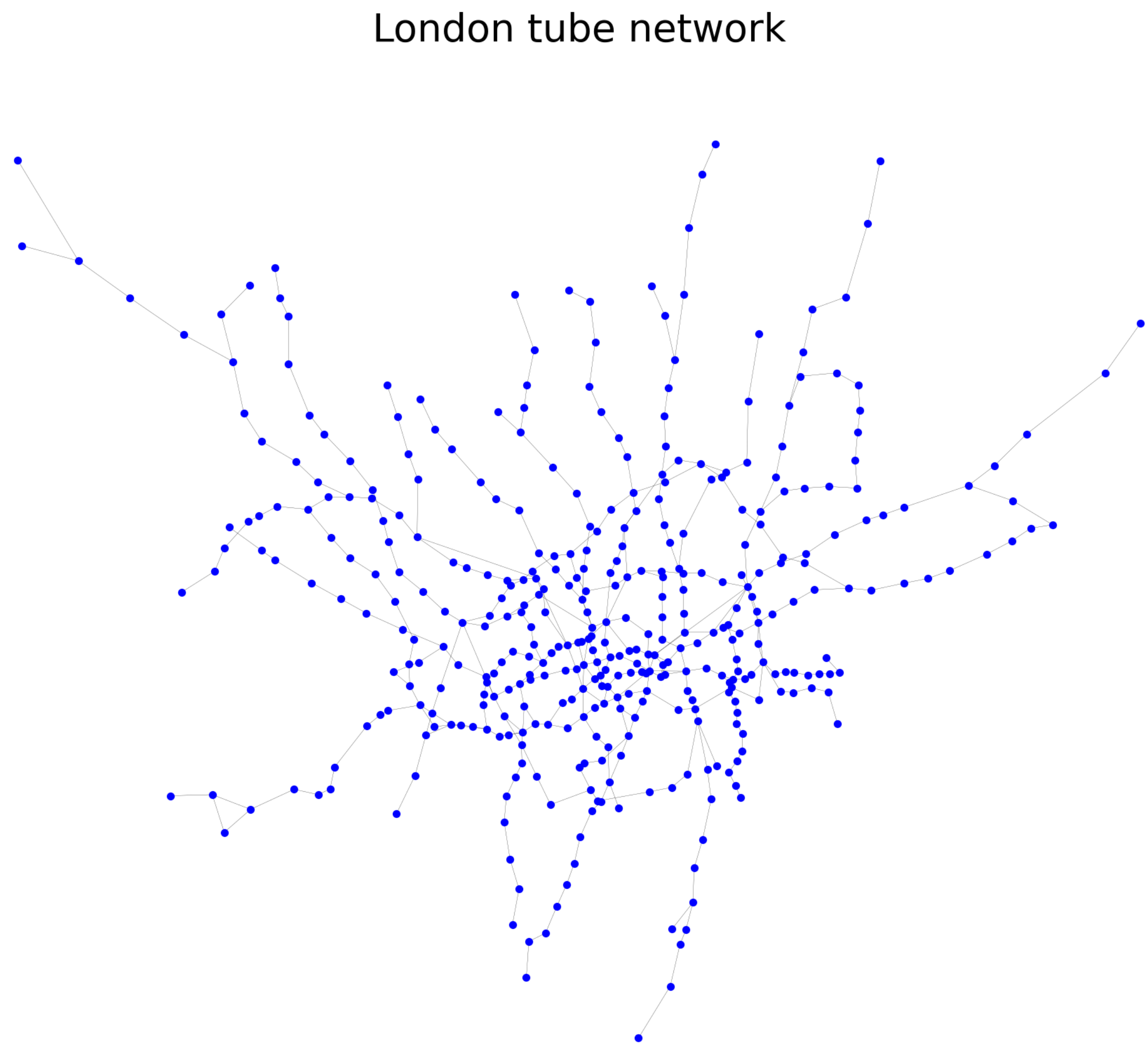

Figure 1. London Tube Network

In [9]:

```
# Compute centralities
degree_centrality = nx.degree_centrality(G)
betweenness_centrality = nx.betweenness_centrality(G)
eigenvector_centrality = nx.eigenvector_centrality_numpy(G)

# Sorting and selecting top 10
sorted_degree = sorted(degree_centrality.items(), key=lambda x: x[1], reverse=True)[:10]
sorted_betweenness = sorted(betweenness_centrality.items(), key=lambda x: x[1], reverse=
sorted_eigenvector = sorted(eigenvector_centrality.items(), key=lambda x: x[1], reverse=

# Create DataFrame from sorted lists
combined_df = pd.DataFrame({
    'Rank': range(1, 11),
    'Degree Centrality Station': [x[0] for x in sorted_degree],
    'Degree Centrality Value': [x[1] for x in sorted_degree],
    'Betweenness Centrality Station': [x[0] for x in sorted_betweenness],
    'Betweenness Centrality Value': [x[1] for x in sorted_betweenness],
    'Eigenvector Centrality Station': [x[0] for x in sorted_eigenvector],
    'Eigenvector Centrality Value': [x[1] for x in sorted_eigenvector]
}).set_index('Rank')
```

```
# Display the DataFrame
display(combined_df)
```

Rank	Degree Centrality Station	Degree Centrality Value	Betweenness Centrality Station	Betweenness Centrality Value	Eigenvector Centrality Station	Eigenvector Centrality Value
1	Stratford	0.0225	Stratford	0.297846	Bank and Monument	0.383725
2	Bank and Monument	0.0200	Bank and Monument	0.290489	Liverpool Street	0.329191
3	Baker Street	0.0175	Liverpool Street	0.270807	Stratford	0.269574
4	King's Cross St. Pancras	0.0175	King's Cross St. Pancras	0.255307	Waterloo	0.249708
5	Liverpool Street	0.0150	Waterloo	0.243921	Moorgate	0.215343
6	West Ham	0.0150	Green Park	0.215835	Green Park	0.197023
7	Canning Town	0.0150	Euston	0.208324	Oxford Circus	0.183441
8	Waterloo	0.0150	Westminster	0.203335	Tower Hill	0.171839
9	Green Park	0.0150	Baker Street	0.191568	Westminster	0.168368
10	Oxford Circus	0.0150	Finchley Road	0.165085	Shadwell	0.159233

Table 1. The first 10 ranked nodes for degree centrality, betweenness centrality and eigenvector centrality

I.2. Impact measures:

Two global measures to evaluate the impact of node removal on the network are:

1. **Global Efficiency**$(\text{Mean Inverse Shortest Path Length}(\langle l^{-1}\rangle))$: This measure provides a way of measuring the overall efficiency of the network by considering the inversed average shortest path length between all pairs of nodes in the network (Latora & Massimo, 2001). For the London Underground network, this means being able to assess the average number of stops a passenger needs to make to get from one stop to another across the network. A high average reverse shortest path length indicates that passengers can reach their destination with fewer transfers or they are use other routes when facing station interruptions, reflecting the high efficiency of the network.

$$\langle l^{-1}\rangle = \frac{1}{N(N-1)} \sum_{j \in V} \sum_{\substack{k \neq j \\ k \in V}} \frac{1}{g_{jk}}$$

1. **Modularity**: This reflects the degree to which a network is compartmentalized into clusters or communities characterized by closely-knit connections within them. A substantial rise in modularity following the elimination of a node may suggest that vital links, which formerly integrated diverse communities and linked distinct areas, have been disrupted. The concept of modularity aids in understanding the network's architectural robustness.

$$Q = \frac{1}{2m} \sum_{ij} \left[A_{ij} - \frac{k_i k_j}{2m} \right] \delta(c_i, c_j)$$

Global Efficientcy

In [10]:
```
# calculate the shortest path between two stations
global_eff = nx.global_efficiency(G)
print(f"Global Efficiency: {global_eff}")
```

```
Global Efficiency: 0.10125619359721513
```

Modularity

In [11]:
```
# greedy_modularity_communities
from networkx.algorithms import community
communities = community.greedy_modularity_communities(G)
modularity = community.modularity(G, communities)
print(f"Modularity: {modularity}")
```

```
Modularity: 0.8302138117924331
```

These measures are not specific to the London Underground; they can be generalized to evaluate the resilience of any network, such as social and biological networks. By comparing the changes in these global measures before and after the removal of critical nodes, researchers can identify the nodes or connections that have the greatest impact on the performance and structure of the network, thus providing guidance for network design and intervention.

I.3. Node removal:

a. non-sequential

1. Prepare the centrality measures and DataFrames

In [12]:
```
# Determine the number of nodes to remove (10)
num_nodes_to_remove = 10
```

2. Remove nodes and calculate metrics

In [13]:
```
G = nx.read_graphml('london_updated.graphml')
G_degree = G.copy()
degree_centrality = nx.degree_centrality(G_degree)

# Sort nodes by degree centrality in descending order
sorted_degree = sorted(degree_centrality.items(), key=lambda x: x[1], reverse=True)
results_degree = [] # Prepare DataFrames to store the results

# Remove nodes based on degree centrality and calculate metrics
for i in range(num_nodes_to_remove):
    node, _ = sorted_degree[i]
    G_degree.remove_node(node)
    global_eff = nx.global_efficiency(G_degree)
    if G_degree.number_of_edges() > 0:
        communities = community.greedy_modularity_communities(G_degree)
        modularity = community.modularity(G_degree, communities)
    else:
        modularity = None
    results_degree.append({'Removing nodes': (i + 1) / num_nodes_to_remove* 100,  # Now
                           'Degree Global Efficiency': global_eff,
                           'Degree Modularity': modularity})


degree = pd.DataFrame(results_degree)
```

In [14]:
```python
G = nx.read_graphml('london_updated.graphml')
G_betweenness = G.copy()
betweenness_centrality = nx.betweenness_centrality(G_betweenness)

# Sort nodes by degree centrality in descending order
sorted_betweenness = sorted(betweenness_centrality.items(), key=lambda x: x[1], reverse=
results_betweenness = []

# Remove nodes based on betweenness centrality and calculate metrics
for i in range(num_nodes_to_remove):
    node, _ = sorted_betweenness[i]
    G_betweenness.remove_node(node)
    global_eff = nx.global_efficiency(G_betweenness)
    if G_betweenness.number_of_edges() > 0:
        communities = community.greedy_modularity_communities(G_betweenness)
        modularity = community.modularity(G_betweenness, communities)
    else:
        modularity = None
    results_betweenness.append({'Removing nodes': (i + 1) / num_nodes_to_remove* 100,  #
                           'Betweenness Global Efficiency': global_eff,
                           'Betweenness Modularity': modularity})


betweenness = pd.DataFrame(results_betweenness)
```

In [15]:
```python
G = nx.read_graphml('london_updated.graphml')
G_eigenvector = G.copy()

try:
    # Increase the number of iterations
    eigenvector_centrality = nx.eigenvector_centrality(G_eigenvector, max_iter=500)
except nx.PowerIterationFailedConvergence:
    print("Eigenvector centrality didn't converge. You might want to check the graph str
sorted_eigenvector = sorted(eigenvector_centrality.items(), key=lambda x: x[1], reverse=
results_eigenvector = []

# Remove nodes based on eigenvector centrality and calculate metrics
for i in range(num_nodes_to_remove):
    node, _ = sorted_eigenvector[i]
    G_eigenvector.remove_node(node)
    global_eff = nx.global_efficiency(G_eigenvector)
    if G_eigenvector.number_of_edges() > 0:
        communities = community.greedy_modularity_communities(G_eigenvector)
        modularity = community.modularity(G_eigenvector, communities)
    else:
        modularity = None
    results_eigenvector.append({'Removing nodes': (i + 1) / num_nodes_to_remove* 100,  #
                           'Eigenvector Global Efficiency': global_eff,
                           'Eigenvector Modularity': modularity})


eigenvector = pd.DataFrame(results_eigenvector)
```

3. Plot the global efficiency and modularity

In [16]:
```python
fig, ax = plt.subplots(1, 2, figsize=(12, 5))

# Plot for Global efficiency
ax[0].plot(degree['Removing nodes'], degree['Degree Global Efficiency'], label='Degree C
ax[0].plot(betweenness['Removing nodes'], betweenness['Betweenness Global Efficiency'],
ax[0].plot(eigenvector['Removing nodes'], eigenvector['Eigenvector Global Efficiency'],
ax[0].set_xlabel('Removing nodes percentage(%)')
ax[0].set_ylabel('Global Efficiency')
ax[0].set_title('Global Efficiency after Node Removal')
```

```
ax[0].legend()

# Plot for Modularity
ax[1].plot(degree['Removing nodes'], degree['Degree Modularity'], label='Degree Centrali
ax[1].plot(betweenness['Removing nodes'], betweenness['Betweenness Modularity'], label='
ax[1].plot(eigenvector['Removing nodes'], eigenvector['Eigenvector Modularity'], label='
ax[1].set_xlabel('Removing nodes percentage(%)')
ax[1].set_ylabel('Modularity')
ax[1].set_title('Modularity after Node Removal')
ax[1].legend()

plt.tight_layout()
plt.show()
```

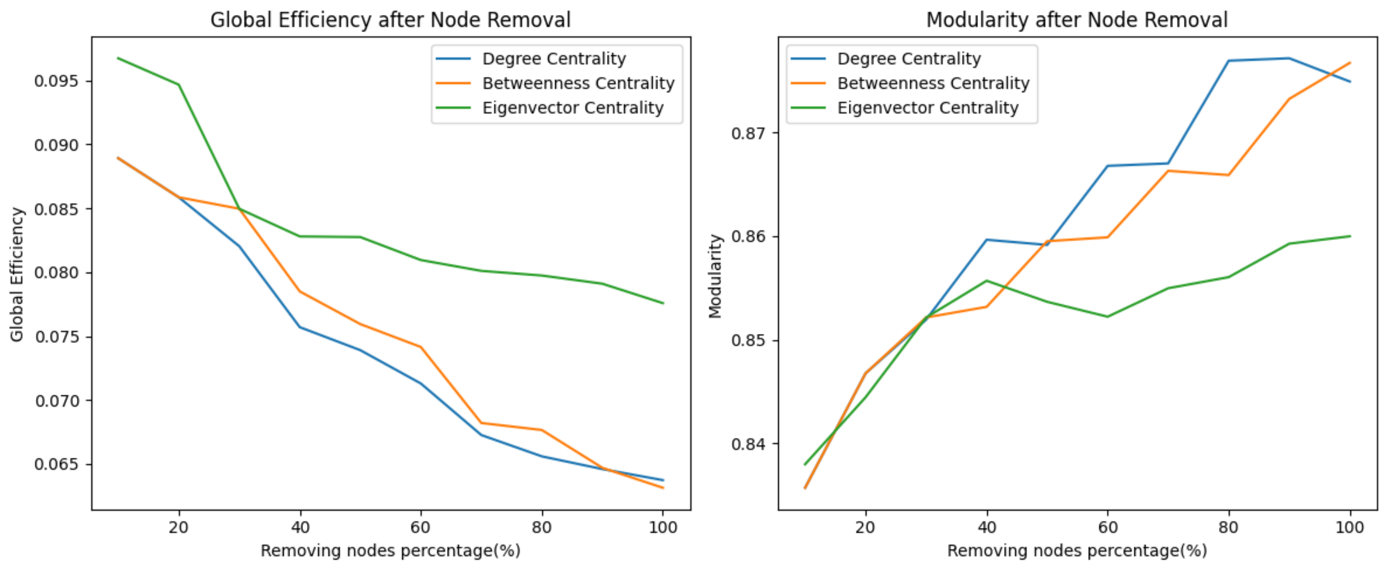

Figure 2. The plot of Non-sequential node removal of two strageties

In [17]:
```
df_nonseq = pd.concat([
    degree[['Degree Global Efficiency', 'Degree Modularity']],
    betweenness[['Betweenness Global Efficiency', 'Betweenness Modularity']],
    eigenvector[['Eigenvector Global Efficiency', 'Eigenvector Modularity']]
], axis=1)

# The 'Removing nodes' column should only be calculated once since the process is the sa
df_nonseq['Removing nodes Percentage'] = df_nonseq.index / 10 * 100

df_nonseq
```

Out[17]:

	Degree Global Efficiency	Degree Modularity	Betweenness Global Efficiency	Betweenness Modularity	Eigenvector Global Efficiency	Eigenvector Modularity	Removing nodes Percentage
0	0.088917	0.835713	0.088917	0.835713	0.096735	0.837980	0.0
1	0.085862	0.846760	0.085862	0.846760	0.094668	0.844449	10.0
2	0.082033	0.851936	0.084963	0.852144	0.084963	0.852144	20.0
3	0.075700	0.859645	0.078498	0.853169	0.082797	0.855688	30.0
4	0.073903	0.859145	0.075942	0.859505	0.082749	0.853656	40.0
5	0.071298	0.866774	0.074152	0.859879	0.080953	0.852218	50.0
6	0.067264	0.867009	0.068206	0.866295	0.080104	0.854965	60.0
7	0.065596	0.876919	0.067660	0.865889	0.079741	0.856036	70.0
8	0.064602	0.877156	0.064700	0.873230	0.079100	0.859264	80.0
9	0.063736	0.874912	0.063139	0.876708	0.077579	0.859977	90.0

Table 2. The Table of Non-sequential node removal of two strageties

b. sequential

1. Prepare the centrality measures and DataFrames

In [18]:
```
num_nodes_to_remove = 10
```

2. Sequentially remove nodes and calculate metrics

In [19]:
```
G = nx.read_graphml('london_updated.graphml')
G_seq_degree = G.copy()
results = []

for i in range(num_nodes_to_remove):
    # calculate the degree centrality of each node
    degree_centrality = nx.degree_centrality(G_seq_degree)

    # sort the dictionary by values in descending order
    sorted_degree = dict(sorted(degree_centrality.items(), key=lambda x: x[1], reverse=T

    # get the first key in the dictionary
    node_to_remove = list(sorted_degree.keys())[0]

    # save the node's name and the degree centrality value of the node to remove
    centrality_value = sorted_degree[node_to_remove]

    # remove the node from the graph
    G_seq_degree.remove_node(node_to_remove)

    # calculate the global efficiency
    global_eff = nx.global_efficiency(G_seq_degree)

    # calculate the modularity
    communities = community.greedy_modularity_communities(G_seq_degree)
    modularity = community.modularity(G_seq_degree, communities)

    results.append({
        'Degree Removed Node': node_to_remove,
        'Degree Centrality': centrality_value,
        'Degree Global Efficiency': global_eff,
        'Degree Modularity': modularity
        })

df_seq_degree = pd.DataFrame(results)
```

In [20]:
```
G = nx.read_graphml('london_updated.graphml')
G_seq_betweenness = G.copy()
results = []

for i in range(num_nodes_to_remove):
    # calculate the betweenness centrality of each node
    betweenness_centrality = nx.betweenness_centrality(G_seq_betweenness)

    # sort the dictionary by values in descending order
    sorted_betweenness = dict(sorted(betweenness_centrality.items(), key=lambda x: x[1],

    # get the first key in the dictionary
    node_to_remove = list(sorted_betweenness.keys())[0]

    # save the node's name and the betweenness centrality value of the node to remove
    centrality_value = sorted_betweenness[node_to_remove]
```

```
    # remove the node from the graph
    G_seq_betweenness.remove_node(node_to_remove)

    # calculate the global efficiency
    global_eff = nx.global_efficiency(G_seq_betweenness)

    # calculate the modularity
    communities = community.greedy_modularity_communities(G_seq_betweenness)
    modularity = community.modularity(G_seq_betweenness, communities)

    results.append({
        'Betweenness Removed Node': node_to_remove,
        'Betweenness Centrality': centrality_value,
        'Betweenness Global Efficiency': global_eff,
        'Betweenness Modularity': modularity
        })

df_seq_betweenness = pd.DataFrame(results)
```

In [21]:

```
G = nx.read_graphml('london_updated.graphml')
G_seq_eigenvector = G.copy()
results = []

for i in range(num_nodes_to_remove):
    try:
        # Calculate eigenvector centrality
        eigenvector_centrality = nx.eigenvector_centrality(G_seq_eigenvector, max_iter=1
        # remove the node with the highest eigenvector centrality
        node_to_remove = max(eigenvector_centrality, key=eigenvector_centrality.get)
        centrality_value = eigenvector_centrality[node_to_remove]

        # remove the node from the graph
        G_seq_eigenvector.remove_node(node_to_remove)

        # calculate global efficiency
        global_eff = nx.global_efficiency(G_seq_eigenvector)

        # calculate modularity
        if G_seq_eigenvector.number_of_nodes() > 0:
            communities = community.greedy_modularity_communities(G_seq_eigenvector)
            modularity = community.modularity(G_seq_eigenvector, communities)
        else:
            modularity = None

        results.append({
            'Eigenvector Removed Node': node_to_remove,
            'Eigenvector Centrality': centrality_value,
            'Eigenvector Global Efficiency': global_eff,
            'Eigenvector Modularity': modularity
        })

    except nx.PowerIterationFailedConvergence as e:
        print(f"Convergence failed at iteration {i + 1}: {e}")
        break  # if the eigenvector centrality doesn't converge, stop the loop

df_seq_eigenvector = pd.DataFrame(results)
```

In [22]:

```
# Every graph has the same number of nodes, so we can use any of them to get the total n
total_nodes = len(G_seq_degree.nodes)  #get the total number of nodes

# combine the three DataFrames
df_seq = pd.concat([df_seq_degree, df_seq_betweenness, df_seq_eigenvector], axis=1)

# calculate the percentage of nodes removed
```

```
df_seq['Removing nodes Percentage'] = df_seq.index /10 * 100

# display the DataFrame
df_seq
```

Out[22]:

	Degree Removed Node	Degree Centrality	Degree Global Efficiency	Degree Modularity	Betweenness Removed Node	Betweenness Centrality	Betweenness Global Efficiency	Betweenness Modularity	Eigenv Rem
0	Stratford	0.022500	0.088917	0.835713	Stratford	0.297846	0.088917	0.835713	Ba Mon
1	Bank and Monument	0.020050	0.085862	0.846760	King's Cross St. Pancras	0.247262	0.084603	0.845011	'
2	Baker Street	0.017588	0.082033	0.851936	Waterloo	0.254180	0.081829	0.856455	St
3	King's Cross St. Pancras	0.017632	0.075700	0.859645	Bank and Monument	0.214651	0.077678	0.860751	Earl's
4	Canning Town	0.015152	0.070396	0.866871	Canada Water	0.244903	0.072832	0.864249	Westr
5	Green Park	0.015190	0.069402	0.870024	West Hampstead	0.456831	0.053210	0.869824	Baker
6	Earl's Court	0.015228	0.067772	0.870985	Earl's Court	0.096182	0.051656	0.871192	King's St. Pa
7	Waterloo	0.012723	0.065936	0.872442	Shepherd's Bush	0.128852	0.045844	0.881066	Ca
8	Oxford Circus	0.012755	0.065069	0.874393	Euston	0.087075	0.041631	0.881247	Tu
9	Willesden Junction	0.012788	0.056748	0.886262	Baker Street	0.098437	0.038164	0.887931	Le S

Table 3. Sequential node removal of two strageties

3. Plot the global efficiency and modularity

In [23]:

```
fig, ax = plt.subplots(1, 2, figsize=(12, 5))

# plot the global efficiency
ax[0].plot(df_seq['Removing nodes Percentage'], df_seq['Degree Global Efficiency'], labe
ax[0].plot(df_seq['Removing nodes Percentage'], df_seq['Betweenness Global Efficiency'],
ax[0].plot(df_seq['Removing nodes Percentage'], df_seq['Eigenvector Global Efficiency'],
ax[0].set_xlabel('Removing nodes percentage(%)')
ax[0].set_ylabel('Global Efficiency')
ax[0].set_title('Global Efficiency After Removing Nodes')
ax[0].legend()

# plot the modularity
ax[1].plot(df_seq['Removing nodes Percentage'], df_seq['Degree Modularity'], label='Degr
ax[1].plot(df_seq['Removing nodes Percentage'], df_seq['Betweenness Modularity'], label=
ax[1].plot(df_seq['Removing nodes Percentage'], df_seq['Eigenvector Modularity'], label=
ax[1].set_xlabel('Removing nodes percentage(%)')
ax[1].set_ylabel('Modularity')
ax[1].set_title('Modularity After Removing Nodes')
ax[1].legend()

plt.show()
```

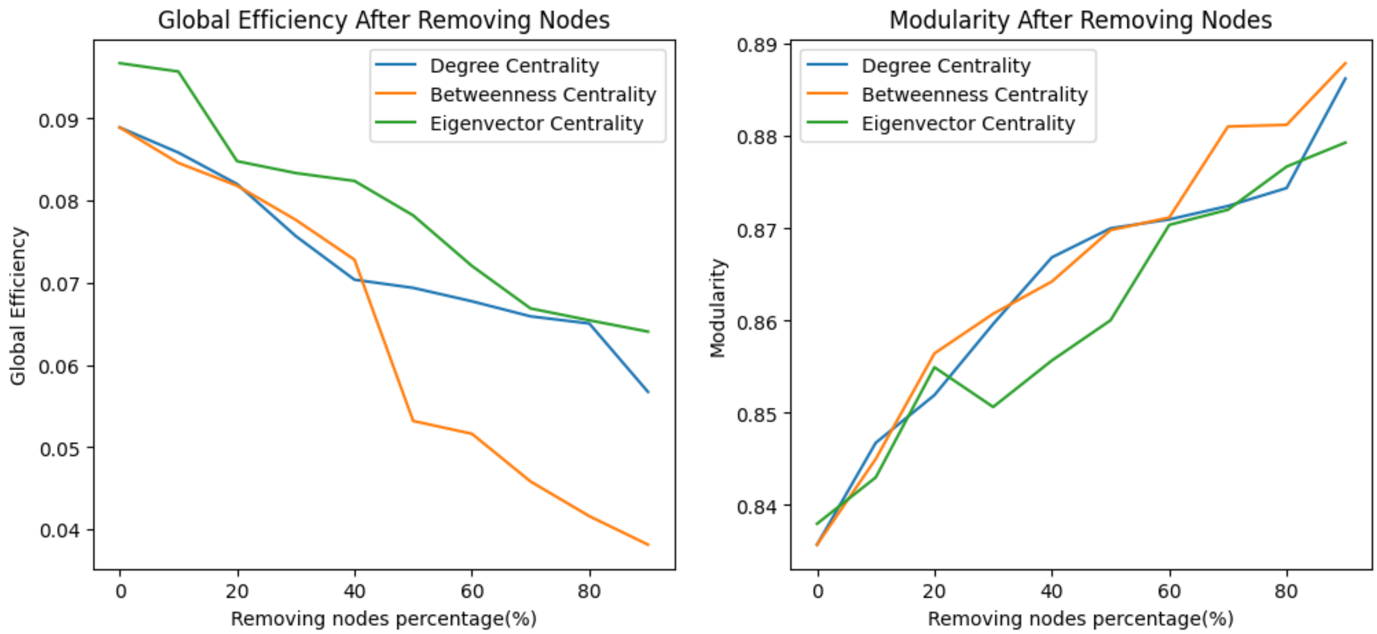

Figure 3. The plot of Sequential node removal of two strageties

Combine the non-sequential and sequential Plot

In [24]:
```python
# Step 1: Rename the columns for non-sequential DataFrames
df_nonseq.columns = [f'Non-Sequential {col}' for col in df_nonseq.columns]

# Step 3: Rename the columns for the sequential DataFrame
df_seq.columns = [f'Sequential {col}' for col in df_seq.columns]

# Step 4: Combine the non-sequential and sequential DataFrames
DF_combined = pd.concat([df_nonseq, df_seq], axis=1)

DF_combined.head()
```

Out[24]:

	Non-Sequential Degree Global Efficiency	Non-Sequential Degree Modularity	Non-Sequential Betweenness Global Efficiency	Non-Sequential Betweenness Modularity	Non-Sequential Eigenvector Global Efficiency	Non-Sequential Eigenvector Modularity	Non-Sequential Removing nodes Percentage	Sequential Degree Removed Node	Seque De Cent
0	0.088917	0.835713	0.088917	0.835713	0.096735	0.837980	0.0	Stratford	0.02
1	0.085862	0.846760	0.085862	0.846760	0.094668	0.844449	10.0	Bank and Monument	0.02
2	0.082033	0.851936	0.084963	0.852144	0.084963	0.852144	20.0	Baker Street	0.01
3	0.075700	0.859645	0.078498	0.853169	0.082797	0.855688	30.0	King's Cross St. Pancras	0.01
4	0.073903	0.859145	0.075942	0.859505	0.082749	0.853656	40.0	Canning Town	0.01

Table 4. The Combined Non-sequential and Sequential node removal of two strageties

In [25]:
```python
fig, ax = plt.subplots(1, 2, figsize=(18, 5))

# plot the global efficiency

# Non-Sequential
ax[0].plot(DF_combined['Non-Sequential Removing nodes Percentage'], DF_combined['Non-Seq
```

```
            label='Non-Sequential Degree Centrality', linestyle='--', color='red', alpha=
ax[0].plot(DF_combined['Non-Sequential Removing nodes Percentage'], DF_combined['Non-Seq
            label='Non-Sequential Betweenness Centrality', linestyle='--', color='green',
ax[0].plot(DF_combined['Non-Sequential Removing nodes Percentage'], DF_combined['Non-Seq
            label='Non-Sequential Eigenvector Centrality', linestyle='--', color='blue',

# Sequential
ax[0].plot(DF_combined['Sequential Removing nodes Percentage'], DF_combined['Sequential
            label='Sequential Degree Centrality', linestyle='-', color='cyan', alpha=0.7)
ax[0].plot(DF_combined['Sequential Removing nodes Percentage'], DF_combined['Sequential
            label='Sequential Betweenness Centrality', linestyle='-', color='olive', alph
ax[0].plot(DF_combined['Sequential Removing nodes Percentage'], DF_combined['Sequential
            label='Sequential Eigenvector Centrality', linestyle='-', color='brown', alph

ax[0].set_xlabel('Removing nodes (%)')
ax[0].set_ylabel('Global Efficiency')
ax[0].set_title('Global Efficiency After Removing Nodes')
ax[0].legend(loc='upper left', bbox_to_anchor=(1, 1))

# plot the modularity

# Non-Sequential
ax[1].plot(DF_combined['Non-Sequential Removing nodes Percentage'], DF_combined['Non-Seq
            label='Non-Sequential Degree Centrality', linestyle='--', color='red', alpha=
ax[1].plot(DF_combined['Non-Sequential Removing nodes Percentage'], DF_combined['Non-Seq
             label='Non-Sequential Betweenness Centrality', linestyle='--', color='green'
ax[1].plot(DF_combined['Non-Sequential Removing nodes Percentage'], DF_combined['Non-Seq
             label='Non-Sequential Eigenvector Centrality', linestyle='--', color='blue',

# Sequential
ax[1].plot(DF_combined['Sequential Removing nodes Percentage'], DF_combined['Sequential
             label='Sequential Degree Centrality', linestyle='-', color='cyan', alpha=0.7
ax[1].plot(DF_combined['Sequential Removing nodes Percentage'], DF_combined['Sequential
             label='Sequential Betweenness Centrality', linestyle='-', color='olive', alp
ax[1].plot(DF_combined['Sequential Removing nodes Percentage'], DF_combined['Sequential
             label='Sequential Eigenvector Centrality', linestyle='-', color='brown', alp

ax[1].set_xlabel('Removing nodes (%)')
ax[1].set_ylabel('Modularity')
ax[1].set_title('Modularity After Removing Nodes')
ax[1].legend(loc='upper left', bbox_to_anchor=(1, 1))

plt.tight_layout()  # Adjust the layout so the legend fits without overlapping the plot
plt.show()
```

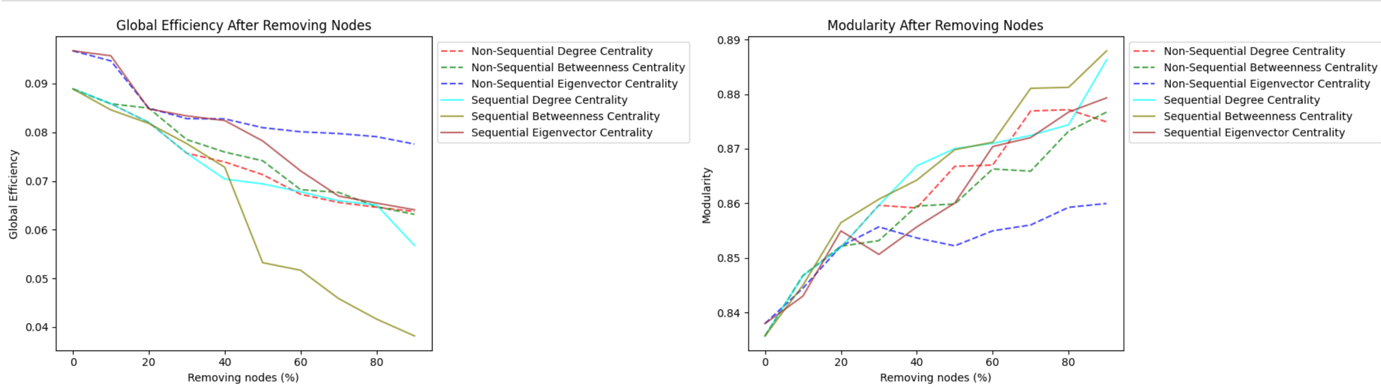

Figure 4. Comparision of Non-sequential and Sequential Node Removal of Two Strageties

Strategies for Studying resilience

Comparing the two strategies, the **sequential removal** is more realistic to the real life situation as the network efficiency should be recaluculated each time after the node removal(attack). As shown in the two plots, they reveal the same situation, where the sequential removal would give greater impact on the network system(global efficiency drop more for sequential than for non-sequential; modularity increase higher for sequential and for non-sequential), as the recalculation would put the existing top one important node into the next round of node removal.

Centrality Measure

It is appeared that eigenvector change the least among the three centrality measures, while betweenness centrality measure is the most sensitive to the node removal for both removal patterns. **Betweenness Centrality**, the larger drop in global efficiency when nodes are removed based on betweenness centrality suggests that these nodes may play a more critical role in the flow through the network. These stations likely act as important bridges in the network, and their removal can significantly impact overall travel efficiency.

Impact measure for assessing damage

In terms of two removal strategies, from the plots shown in figure 3, both work very well, but in the case of examing the impact of node removal, **global efficiency** is a better choice compare to modularity as the modularity has this up and down changing pattern which could lead to confusion in the practice (global efficiency is changing in one direction).

II. Flows: weighted network

II.1. Define flow and the Top 10 nodes

In [26]:
```
G = nx.read_graphml('london_updated.graphml')
G_weighted = G.copy()
# To check edge attributes:
#G.edges(data=True)
```

To consider the weighted network in the Underground, a weighted parameter "flows" and "length" can be used for the weighted network. I am going to use both since taking passengers into consideration can both relates to the traveling time (length/distance between the stations) and the commuting population density (flows). They are both important parameters when stations are removed.

Compare the weighted results to the the topological results, the top 10 stations are semi-different.

In [27]:
```
# computing the degree centrality
weighted_degree_centrality = {}
for node in G_weighted.nodes():
    weighted_degree = sum(weight for _, _, weight in G_weighted.edges(node, data='length
    weighted_degree_centrality[node] = weighted_degree

# computing the betweenness centrality
weighted_betweenness_centrality = nx.betweenness_centrality(G_weighted, weight='length')

# computing the eigenvector centrality
weighted_eigenvector_centrality = nx.eigenvector_centrality_numpy(G_weighted, weight='le
```

In [28]:
```
# Sorting and selecting top 10
sorted_degree = sorted(weighted_degree_centrality.items(), key=lambda x: x[1], reverse=T
sorted_betweenness = sorted(weighted_betweenness_centrality.items(), key=lambda x: x[1],
sorted_eigenvector = sorted(weighted_eigenvector_centrality.items(), key=lambda x: x[1],

# Create DataFrame from sorted lists
combined_weighted_df = pd.DataFrame({
```

```
    'Rank': range(1, 11),
    'Degree Centrality Station': [x[0] for x in sorted_degree],
    'Degree Centrality Value': [x[1] for x in sorted_degree],
    'Betweenness Centrality Station': [x[0] for x in sorted_betweenness],
    'Betweenness Centrality Value': [x[1] for x in sorted_betweenness],
    'Eigenvector Centrality Station': [x[0] for x in sorted_eigenvector],
    'Eigenvector Centrality Value': [x[1] for x in sorted_eigenvector]
}).set_index('Rank')

# Display the DataFrame
display(combined_weighted_df)
```

Rank	Degree Centrality Station	Degree Centrality Value	Betweenness Centrality Station	Betweenness Centrality Value	Eigenvector Centrality Station	Eigenvector Centrality Value
1	Stratford	18809.755024	Bank and Monument	0.221504	Wembley Park	0.640161
2	Wembley Park	13955.803937	King's Cross St. Pancras	0.209674	Finchley Road	0.635998
3	Chalfont & Latimer	13214.405571	Stratford	0.182494	Baker Street	0.287363
4	Finchley Road	12201.911381	Baker Street	0.164248	Kingsbury	0.210939
5	Liverpool Street	12070.083236	Oxford Circus	0.157306	Neasden	0.168126
6	Willesden Junction	11100.258775	Euston	0.155138	Preston Road	0.104587
7	Baker Street	10613.160373	Earl's Court	0.143521	St. John's Wood	0.073800
8	King's Cross St. Pancras	10453.003671	Shadwell	0.139449	Bond Street	0.057430
9	Bank and Monument	9383.512648	Waterloo	0.130213	West Hampstead	0.054723
10	Heathrow Terminals 2 & 3	9054.318001	South Kensington	0.129110	Swiss Cottage	0.053379

Table 5. Top 10 Stations for a Weighted(length/distance) Tube Network

In [29]:

```
# computing the degree centrality
weighted_degree_centrality = {}
for node in G_weighted.nodes():
    weighted_degree = sum(weight for _, _, weight in G_weighted.edges(node, data='flows'
    weighted_degree_centrality[node] = weighted_degree

# computing the betweenness centrality
weighted_betweenness_centrality = nx.betweenness_centrality(G_weighted, weight='flows')

# computing the eigenvector centrality
weighted_eigenvector_centrality = nx.eigenvector_centrality_numpy(G_weighted, weight='fl
```

In [30]:

```
# Sorting and selecting top 10
sorted_degree = sorted(weighted_degree_centrality.items(), key=lambda x: x[1], reverse=T
sorted_betweenness = sorted(weighted_betweenness_centrality.items(), key=lambda x: x[1],
sorted_eigenvector = sorted(weighted_eigenvector_centrality.items(), key=lambda x: x[1],

# Create DataFrame from sorted lists
combined_weighted_df = pd.DataFrame({
    'Rank': range(1, 11),
    'Degree Centrality Station': [x[0] for x in sorted_degree],
```

```python
    'Degree Centrality Value': [x[1] for x in sorted_degree],
    'Betweenness Centrality Station': [x[0] for x in sorted_betweenness],
    'Betweenness Centrality Value': [x[1] for x in sorted_betweenness],
    'Eigenvector Centrality Station': [x[0] for x in sorted_eigenvector],
    'Eigenvector Centrality Value': [x[1] for x in sorted_eigenvector]
}).set_index('Rank')

# Display the DataFrame
display(combined_weighted_df)
```

Rank	Degree Centrality Station	Degree Centrality Value	Betweenness Centrality Station	Betweenness Centrality Value	Eigenvector Centrality Station	Eigenvector Centrality Value
1	Green Park	713696	West Hampstead	0.396617	Green Park	0.527545
2	Bank and Monument	583541	Gospel Oak	0.295238	Westminster	0.495329
3	Waterloo	570862	Finchley Road & Frognal	0.285821	Waterloo	0.416148
4	King's Cross St. Pancras	483565	Hampstead Heath	0.284978	Bank and Monument	0.286512
5	Westminster	461343	Willesden Junction	0.267440	Victoria	0.279919
6	Liverpool Street	437335	Stratford	0.261758	Oxford Circus	0.202460
7	Victoria	383024	Brondesbury	0.243142	Bond Street	0.194256
8	Euston	374576	Brondesbury Park	0.241667	Liverpool Street	0.143603
9	Stratford	366679	Kensal Rise	0.240204	Southwark	0.076285
10	Oxford Circus	325756	Baker Street	0.169160	Sloane Square	0.075176

Table 6. Top 10 Stations for a Weighted (flows) Tube Network

Comparing the two tables above, it shows that when taking different weights into consideration can lead to quite different results.

II.2.

Global Efficiency and modularity are enough for examing the weighted network as the former one can be used for accessing "distance(length)" as the weight while the latter one be used for accessing "population(flows)" as the weight.

The **weighted global efficiency** is a measure of efficiency in a network, which is often used to quantify the efficiency of information or material flows between nodes in the network. The weights (weight) represent distances (distance) in this context, which means higher weights indicate larger distances, and hence, lower efficiency.

Definition: Given a weighted graph, the global efficiency E_{glob} is defined as the average inverse weighted shortest path length between each pair of nodes. For any two nodes u and v, the efficiency $e(u,v)$ is given by $e(u,v) = \frac{1}{d(u,v)}$, where $d(u,v)$ is the shortest weighted path length between nodes (weight).

Formula:

$$E_{glob} = \frac{1}{N(N-1)} \sum_{u,v \in V, u \neq v} \frac{1}{d(u,v)}$$

Here, N is the total number of nodes, V is the set of nodes, and $d(u,v)$ is the weighted shortest path length between the nodes u and v (considering weights as distances).

In [31]:

```
G_weighted = G.copy()

def weighted_global_efficiency(G, node_all=None):
    """计算带权重的全局效率"""
    n = len(G)
    if n <= 1:
        return 0  # 如果图中只有一个或没有节点，全局效率为0

    if node_all is None:
        node_all = n

    dist_sum = 0
    path_count = 0
    # 使用 Dijkstra 算法计算所有节点对之间的最短路径长度
    for lengths in nx.all_pairs_dijkstra_path_length(G, weight='distance'):
        for target, dist in lengths[1].items():
            if  dist >0:
                # 只有当源节点和目标节点不同时，才计算效率
                dist_sum += 1 / dist
                path_count += 1

    # 如果有有效的路径对，计算平均全局效率；否则返回0
    if node_all > 1:
        return dist_sum / ((node_all * (node_all - 1)) / 2)
    else:
        return 0

# 调用函数计算全局效率
original_total_nodes = len(G_weighted)
global_eff = weighted_global_efficiency(G_weighted, node_all=original_total_nodes)
print("Weighted Global Efficiency:", global_eff)
```

```
Weighted Global Efficiency: 0.20251238719443027
```

For **weighted modularity**, a high modularity value indicates that the network can be clearly classified into modules or communities consisting of densely connected nodes internally. When calculating modularity, the weight parameter is used to indicate the importance of edges, such as traffic, capacity, etc., and in this case population flows, which helps to determine which connections are more critical to the community structure.

In [32]:

```
G_weighted = G.copy()

communities = community.greedy_modularity_communities(G_weighted, weight='flows')

modularity = community.modularity(G_weighted, communities, weight='flows')
print(f"Modularity is: {modularity}")
```

```
Modularity is: 0.7321343579048248
```

II.3. Remove the 3 highest ranked nodes

Using betweenness centrality:

In [33]:

```
G_BC = G.copy()
G_BC.nodes(data=True)
node_name = "Bank and Monument"
```

```
G_BC.remove_node(node_name)
# calculate the global efficiency
global_eff = weighted_global_efficiency(G_BC, node_all=original_total_nodes)
# calculate the modularity
communities = community.greedy_modularity_communities(G_BC, weight='flows')
modularity = community.modularity(G_BC, communities, weight='flows')
print({'Removed Node': node_name,
                'Global Efficiency': global_eff,
                'Modularity': modularity})
```

```
{'Removed Node': 'Bank and Monument', 'Global Efficiency': 0.19250457694682765, 'Modular
ity': 0.7686870265089242}
```

In [34]:
```
G_nonseq_removal_weighted = G.copy()
```

In [35]:
```
betweenness = nx.betweenness_centrality(G, weight='length')

# 将结果转换为DataFrame
weighted_centrality_df = pd.DataFrame(list(betweenness.items()), columns=['Station', 'be

# 对介数中心性进行降序排序，并取前三个站点的名称
top_3_betweenness = weighted_centrality_df.sort_values(by='betweenness_centrality', asce
```

In [36]:
```
G_nonseq_removal_weighted = G.copy()
# prepare the dataframe to store the results
results = []

remove_list = top_3_betweenness

for node_name in remove_list:
    # remove the node from the graph
    G_nonseq_removal_weighted.remove_node(node_name)
    # calculate the global efficiency
    global_eff = weighted_global_efficiency(G_nonseq_removal_weighted, node_all=original
    # calculate the modularity
    communities = community.greedy_modularity_communities(G_nonseq_removal_weighted, wei
    modularity = community.modularity(G_nonseq_removal_weighted, communities, weight='fl
    # save the results to the dataframe
    results.append({'Removed Node': node_name,
                    'Global Efficiency': global_eff,
                    'Modularity': modularity})
df_nonseq_betweenness_weighted = pd.DataFrame(results)
```

In [37]:
```
G_seq_removal_weighted = G.copy()
results = []

#remove the top one betweenness centrality node
for i in range(3):
    # calculate the betweenness centrality
    betweenness_centrality = nx.betweenness_centrality(G_seq_removal_weighted, weight='l
    # sort the dictionary by values in descending order
    sorted_betweenness = dict(sorted(betweenness_centrality.items(), key=lambda x: x[1],
    # get the first key in the dictionary
    node_to_remove = list(sorted_betweenness.keys())[0]
    # remove the node from the graph
    G_seq_removal_weighted.remove_node(node_to_remove)
    # calculate the global efficiency
    global_eff = weighted_global_efficiency(G_seq_removal_weighted, node_all=original_to
    # calculate the modularity
    communities = community.greedy_modularity_communities(G_seq_removal_weighted, weight
    modularity = community.modularity(G_seq_removal_weighted, communities, weight='flows
    results.append({'Removed Node': node_to_remove,
                    'Global Efficiency': global_eff,
                    'Modularity': modularity})
df_seq_betweenness_weighted = pd.DataFrame(results)
```

In [38]:
```
#combine the dataframe
DF_combined_weighted = pd.concat([df_nonseq_betweenness_weighted, df_seq_betweenness_wei
new_columns = [
    'Removed Node Nonsequential', 'Global Efficiency Nonsequential', 'Modularity Nonsequ
    'Removed Node Sequential', 'Global Efficiency Sequential', 'Modularity Sequential'
]
DF_combined_weighted.columns = new_columns
DF_combined_weighted
```

Out[38]:

	Removed Node Nonsequential	Global Efficiency Nonsequential	Modularity Nonsequential	Removed Node Sequential	Global Efficiency Sequential	Modularity Sequential
0	Bank and Monument	0.192505	0.768687	Bank and Monument	0.192505	0.768687
1	King's Cross St. Pancras	0.177976	0.787755	King's Cross St. Pancras	0.177976	0.787755
2	Stratford	0.158177	0.798175	Canada Water	0.163636	0.790404

Table 7. Top 3 Stations for a Weighted Tube Network

Non sequential seems to be slightly more damaged than the sequential way after the third removal while surprisingly stay the same in the first two removals. Therefore, Bank and Monument Station and St.Pancras Station and Stratford Station are the three most important stations in the tube network.

Part 2: Spatial Interaction models

III. Models and calibration

III.1. Model Definition

There is a famility of spatial interaction model that shows different sptial interaction patterns:

1. The Unconstrained Model The model describes the proportionality of the product of the mass of the origin and destination and the inversley proportional to the distance between them.

$$T_{ij} = k\frac{O_i^{\alpha} D_j^{\gamma}}{d_{ij}^{\beta}} \tag{1}$$

Wilson's version (1971) of the family of gravity models:

$$T_{ij} = kO_i^{\alpha} D_j^{\gamma} d_{ij}^{-\beta} \tag{2}$$

T_{ij} : The flows from the origin station i to the destination station j.

O_i: The origion station i 's population.

D_j: The destination station j 's attractivness(jobs).

d_{ij}: The cost or distance from station i to j.

K: The model parameters, or calibration constant.

β: distance decay parameter.

2. The Singly-Constrained Model

It is usually used in the constomer transport behavior analysis. Constrain one parameter.

Production (orign) Constrained Spatial Interaction Model

$$T_{ij} = A_i O_i D_j \exp(-\beta c_{ij}) \quad \text{subject to} \quad \sum_{j=1}^{m} T_{ij} = O_i \tag{3}$$

A_i:Attractiveness of the destination.

Attraction (destination) Constrained Spatial Interaction Model

$$T_{ij} = B_j O_i D_j \exp(-\beta c_{ij}) \quad \text{subject to} \quad \sum_{i=1}^{n} T_{ij} = D_j \tag{4}$$

B_j: A scaling paprameter for alignment of origin i.

3. The Doubly-Costrained Model

It is a comprehensive model udapted from the singly-constrained model, where it sets both the origins and the destinations in the model as the fixed constrains.

$$T_{ij} = A_i O_i B_j D_j \exp(-\beta c_{ij}) \quad \text{subject to} \quad \sum_{j=1}^{m} T_{ij} = O_i \quad \text{and} \quad \sum_{i=1}^{n} T_{ij} = D_j \tag{5}$$

III.2. Model Calibration

Use singly-constrained model because we interested in the flows depend on the population(fixed) and the jobs/attractiveness of the places(not fixed) in the destination would be better suited for the scenario testing where one parameter is fixed and the other is not.

We need to make the model as accurately predict the actually flow as possible.

The rest of the notebook is some thinking process as I don't really understand the concept, so the rest of the codes are just the prove that I tried...

In [3]:
```python
# import all the necessary libraries
import os
import pandas as pd
import numpy as np
from scipy.optimize import minimize
from scipy.special import expit
import matplotlib.pyplot as plt
import geopandas as gpd
import seaborn as sns
import folium
import statsmodels.api as sm
import scipy.stats
from math import sqrt
import statsmodels.formula.api as smf
from scipy.stats import norm
import networkx as nx
```

```
C:\Users\lenovo\AppData\Local\Temp\ipykernel_15608\2158460682.py:3: DeprecationWarning:
Pyarrow will become a required dependency of pandas in the next major release of pandas
(pandas 3.0),
(to allow more performant data types, such as the Arrow string type, and better interope
rability with other libraries)
but was not found to be installed on your system.
If this would cause problems for you,
```

```
please provide us feedback at https://github.com/pandas-dev/pandas/issues/54466

  import pandas as pd
```

In [4]:
```
df = pd.read_csv('London_flows.csv')
df.head()
```

Out[4]:

	station_origin	station_destination	flows	population	jobs	distance
0	Abbey Road	Bank and Monument	0	599	78549	8131.525097
1	Abbey Road	Beckton	1	599	442	8510.121774
2	Abbey Road	Blackwall	3	599	665	3775.448872
3	Abbey Road	Canary Wharf	1	599	58772	5086.514220
4	Abbey Road	Canning Town	37	599	15428	2228.923167

Table 8. London Flows

Data Preprocessing

In [5]:
```
#(drop the rows that is zero and in this case drop Battersea Park as it has no data for
df_drop = df[df['flows'] != 0]
df_drop.head()
```

Out[5]:

	station_origin	station_destination	flows	population	jobs	distance
1	Abbey Road	Beckton	1	599	442	8510.121774
2	Abbey Road	Blackwall	3	599	665	3775.448872
3	Abbey Road	Canary Wharf	1	599	58772	5086.514220
4	Abbey Road	Canning Town	37	599	15428	2228.923167
5	Abbey Road	Crossharbour	1	599	1208	6686.475560

Table 9. London Flows Revised

Log

In [6]:
```
df_drop['flow'] = df_drop['flows']
df_drop['log_flow'] = np.log(df_drop['flow'])
df_drop['dist'] = df_drop['distance'] + 1e-6 + 1
df_drop['log_dist'] = np.log(df_drop['dist'])
df_drop['log_jobs'] = np.log(df_drop['jobs'])
df_origin = df_drop.copy()
df_drop.head()
```

```
C:\Users\lenovo\AppData\Local\Temp\ipykernel_15608\4050372465.py:1: SettingWithCopyWarni
ng:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_
guide/indexing.html#returning-a-view-versus-a-copy
  df_drop['flow'] = df_drop['flows']
C:\Users\lenovo\AppData\Local\Temp\ipykernel_15608\4050372465.py:2: SettingWithCopyWarni
ng:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_
```

```
guide/indexing.html#returning-a-view-versus-a-copy
  df_drop['log_flow'] = np.log(df_drop['flow'])
C:\Users\lenovo\AppData\Local\Temp\ipykernel_15608\4050372465.py:3: SettingWithCopyWarni
ng:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_
guide/indexing.html#returning-a-view-versus-a-copy
  df_drop['dist'] = df_drop['distance'] + 1e-6 + 1
C:\Users\lenovo\AppData\Local\Temp\ipykernel_15608\4050372465.py:4: SettingWithCopyWarni
ng:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_
guide/indexing.html#returning-a-view-versus-a-copy
  df_drop['log_dist'] = np.log(df_drop['dist'])
C:\Users\lenovo\AppData\Local\Temp\ipykernel_15608\4050372465.py:5: SettingWithCopyWarni
ng:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_
guide/indexing.html#returning-a-view-versus-a-copy
  df_drop['log_jobs'] = np.log(df_drop['jobs'])
```

Out[6]:

	station_origin	station_destination	flows	population	jobs	distance	flow	log_flow	dist	log_
1	Abbey Road	Beckton	1	599	442	8510.121774	1	0.000000	8511.121775	9.04
2	Abbey Road	Blackwall	3	599	665	3775.448872	3	1.098612	3776.448873	8.23
3	Abbey Road	Canary Wharf	1	599	58772	5086.514220	1	0.000000	5087.514221	8.53
4	Abbey Road	Canning Town	37	599	15428	2228.923167	37	3.610918	2229.923168	7.70
5	Abbey Road	Crossharbour	1	599	1208	6686.475560	1	0.000000	6687.475561	8.80

Table 10. London Flows Logged

In [7]:
```
print("variance of flows: ", df_origin['flows'].var())
print("variance of log_flow: ", df_origin['log_flow'].var())
```

```
variance of flows:  23804.54155601981
variance of log_flow:  2.55093809831381
```

In [9]:
```
fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(16, 6))  # 1行2列的子图，整体图形大小为16

axes[0].hist(df_origin['flows'], bins=20, edgecolor='white', linewidth=0.5)
axes[0].set_title('Histogram of observed flows')
axes[0].set_yscale('log')
axes[0].set_xlabel('Population flows')
axes[0].set_ylabel('Frequency')
axes[0].set_xlim(left=1)

axes[1].hist(df_origin['log_flow'], bins=20, edgecolor='white', linewidth=0.5)
axes[1].set_title('Histogram of observed log_flow')
axes[1].set_yscale('log')
axes[1].set_xlabel('Population flows (after log transformation)')
axes[1].set_ylabel('Frequency')
axes[1].set_xlim(left=0)

plt.tight_layout()
plt.show()
```

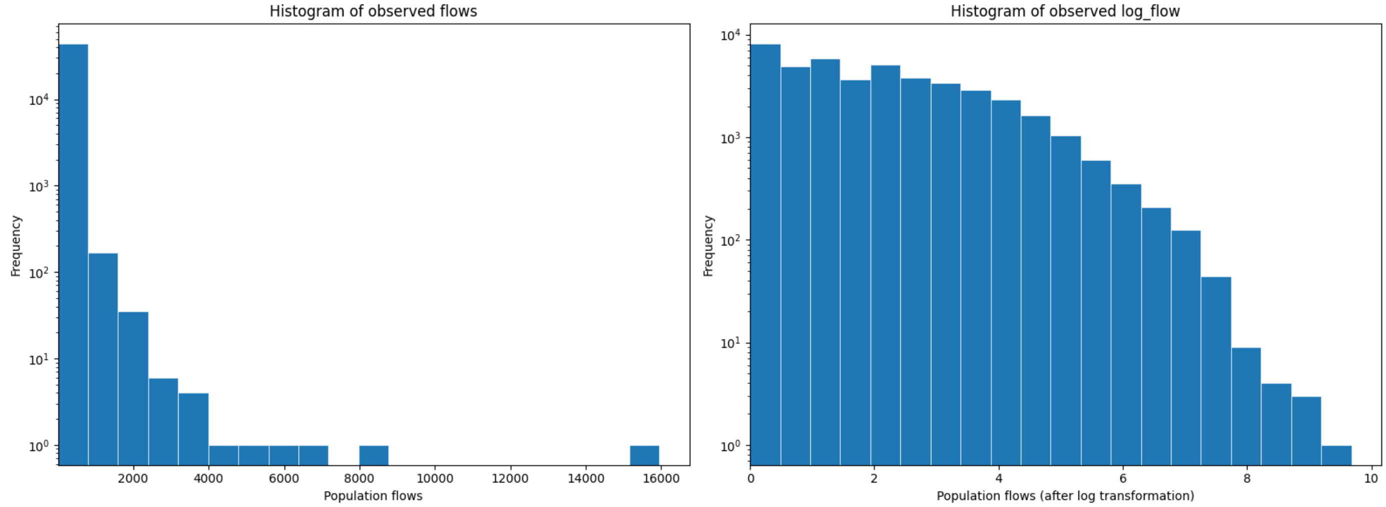

Figure 5. Traffic Flows Histograms of flows and logged flows

The histograms above shows that the flows is like a power-law distribution.

Plot the log-log Flow Plot

In [11]:
```
#subset the dataframe to the flows we want
cdata_flows = df_origin[["flows", "distance"]]
#remove all 0 values (logarithms can't deal with 0 values)
cdata_flows = cdata_flows[(cdata_flows!=0).all(1)]

#extract the x and y converting to log
x = np.log(cdata_flows["flows"])
y = np.log(cdata_flows["distance"])

#create the subplot
fig, ax = plt.subplots(figsize = (10,10))

#plot the results along with the line of best fit
sns.regplot(x=x, y=y, marker="+", ax=ax)

# set the title
ax.set_title("Log-Log plot of flows vs distance")
ax.set_xlabel("log(distance)")
ax.set_ylabel("log(flows)")
plt.show()
```

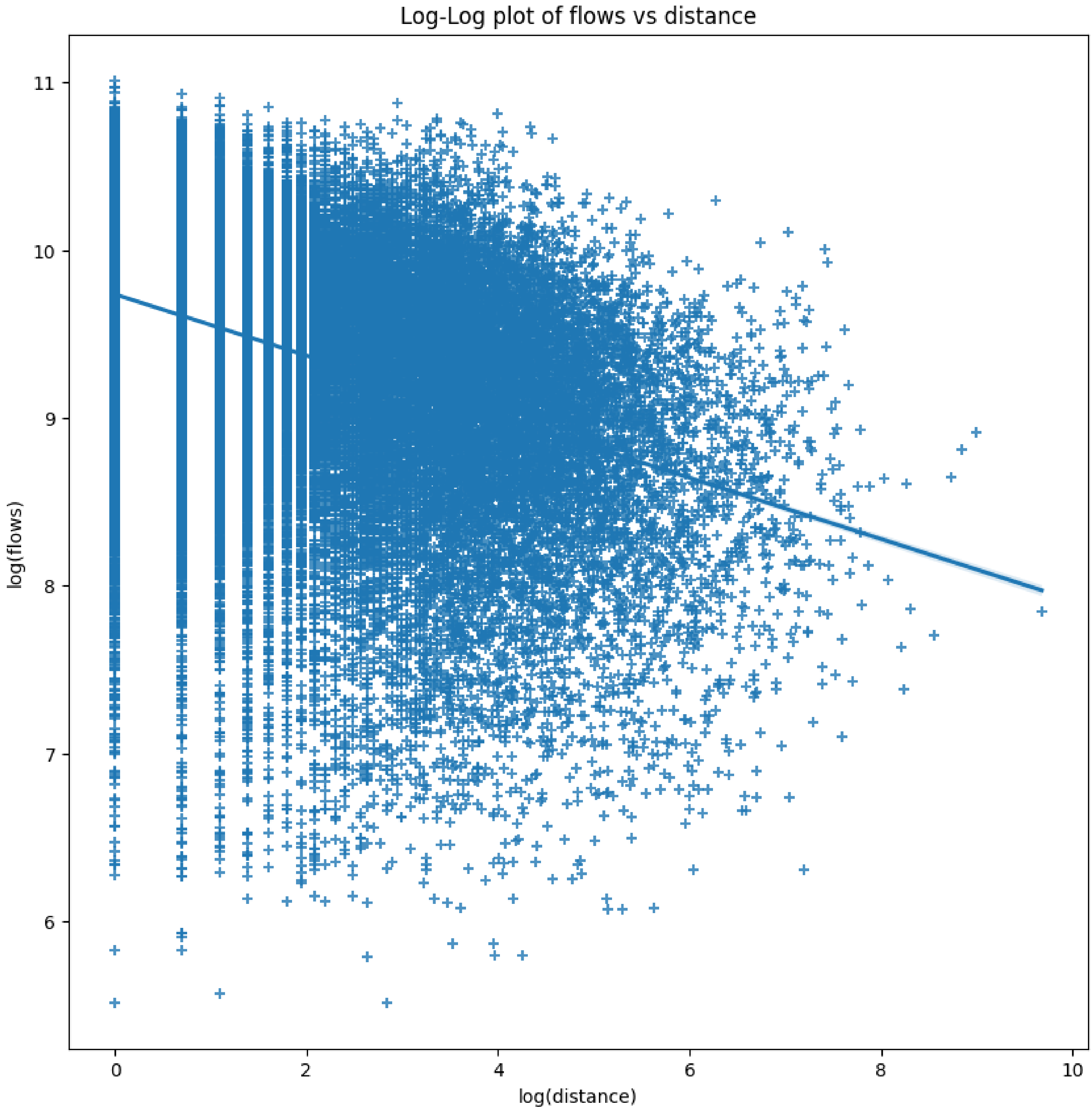

Figure 5. Log-log Plot of Flows vs Distance

In [20]:
```
#create the formula (the "-1" indicates no intercept in the regression model).
formula = 'flow ~ station_origin + log_jobs + log_dist-1'
#run a production constrained sim
prodSim = smf.glm(formula = formula, data=df_drop, family=sm.families.Poisson()).fit()
#let's have a look at it's summary
print(prodSim.summary())
```

```
                 Generalized Linear Model Regression Results
==============================================================================
Dep. Variable:                   flow   No. Observations:                43952
Model:                            GLM   Df Residuals:                    43552
Model Family:                 Poisson   Df Model:                          399
Link Function:                    Log   Scale:                          1.0000
Method:                          IRLS   Log-Likelihood:            -1.0249e+06
Date:                Mon, 29 Apr 2024   Deviance:                   1.8776e+06
Time:                        17:19:15   Pearson chi2:                 2.98e+06
No. Iterations:                    10   Pseudo R-squ. (CS):              1.000
Covariance Type:            nonrobust
```

```
==========================================================================================
=======================
                                                 coef    std err          z      P>|z|
     [0.025      0.975]
------------------------------------------------------------------------------------------
-----------------------
station_origin[Abbey Road]                    -0.2183      0.042     -5.235      0.000
     -0.300      -0.137
station_origin[Acton Central]                  0.8557      0.030     28.646      0.000
      0.797       0.914
station_origin[Acton Town]                     0.1308      0.019      7.019      0.000
      0.094       0.167
station_origin[Aldgate]                       -0.3370      0.021    -16.424      0.000
     -0.377      -0.297
station_origin[Aldgate East]                  -0.3595      0.020    -18.223      0.000
     -0.398      -0.321
station_origin[All Saints]                    -0.2626      0.038     -6.959      0.000
     -0.337      -0.189
station_origin[Alperton]                      -0.3821      0.026    -14.435      0.000
     -0.434      -0.330
station_origin[Amersham]                      -0.3385      0.031    -11.069      0.000
     -0.398      -0.279
station_origin[Anerley]                        0.7933      0.040     19.645      0.000
      0.714       0.872
station_origin[Angel]                         -0.1295      0.018     -7.329      0.000
     -0.164      -0.095
station_origin[Archway]                        0.4322      0.016     27.014      0.000
      0.401       0.464
station_origin[Arnos Grove]                   -0.0183      0.020     -0.904      0.366
     -0.058       0.021
station_origin[Arsenal]                       -0.5824      0.023    -25.050      0.000
     -0.628      -0.537
station_origin[Baker Street]                   0.4410      0.014     30.613      0.000
      0.413       0.469
station_origin[Balham]                         1.0167      0.014     74.948      0.000
      0.990       1.043
station_origin[Bank and Monument]              1.6668      0.010    163.372      0.000
      1.647       1.687
station_origin[Barbican]                      -1.2140      0.030    -40.906      0.000
     -1.272      -1.156
station_origin[Barking]                        0.9252      0.014     64.813      0.000
      0.897       0.953
station_origin[Barkingside]                   -0.6337      0.029    -21.664      0.000
     -0.691      -0.576
station_origin[Barons Court]                   0.0426      0.018      2.369      0.018
      0.007       0.078
station_origin[Bayswater]                     -0.6274      0.027    -23.500      0.000
     -0.680      -0.575
station_origin[Beckton]                        0.7600      0.030     24.972      0.000
      0.700       0.820
station_origin[Beckton Park]                  -0.5616      0.059     -9.503      0.000
     -0.677      -0.446
station_origin[Becontree]                     -0.1522      0.023     -6.710      0.000
     -0.197      -0.108
station_origin[Belsize Park]                  -0.2653      0.021    -12.607      0.000
     -0.307      -0.224
station_origin[Bermondsey]                     0.1570      0.016      9.524      0.000
      0.125       0.189
station_origin[Bethnal Green]                  0.4184      0.015     28.042      0.000
      0.389       0.448
station_origin[Blackfriars]                    0.2024      0.015     13.954      0.000
      0.174       0.231
station_origin[Blackhorse Road]                0.9236      0.014     65.661      0.000
      0.896       0.951
station_origin[Blackwall]                      0.0640      0.034      1.908      0.056
     -0.002       0.130
```

station_origin[Bond Street]	-0.8295	0.024	-34.946	0.000	-0.876	-0.783
station_origin[Borough]	-1.1933	0.029	-41.748	0.000	-1.249	-1.137
station_origin[Boston Manor]	-0.7942	0.031	-26.027	0.000	-0.854	-0.734
station_origin[Bounds Green]	0.3028	0.018	17.284	0.000	0.268	0.337
station_origin[Bow Church]	0.6831	0.026	26.272	0.000	0.632	0.734
station_origin[Bow Road]	0.0892	0.018	4.937	0.000	0.054	0.125
station_origin[Brent Cross]	-0.6786	0.027	-24.747	0.000	-0.732	-0.625
station_origin[Brentwood]	2.3532	0.023	100.216	0.000	2.307	2.399
station_origin[Brixton]	1.7510	0.011	159.616	0.000	1.729	1.772
station_origin[Brockley]	1.6643	0.022	74.339	0.000	1.620	1.708
station_origin[Bromley-by-Bow]	-0.3710	0.022	-17.113	0.000	-0.413	-0.328
station_origin[Brondesbury]	1.0954	0.024	45.600	0.000	1.048	1.142
station_origin[Brondesbury Park]	0.0953	0.039	2.444	0.015	0.019	0.172
station_origin[Bruce Grove]	1.2098	0.033	36.276	0.000	1.144	1.275
station_origin[Buckhurst Hill]	-0.3926	0.027	-14.777	0.000	-0.445	-0.341
station_origin[Burnt Oak]	0.0914	0.020	4.484	0.000	0.051	0.131
station_origin[Bush Hill Park]	1.7340	0.028	61.446	0.000	1.679	1.789
station_origin[Bushey]	-0.2540	0.065	-3.882	0.000	-0.382	-0.126
station_origin[Caledonian Road]	-0.3757	0.021	-18.154	0.000	-0.416	-0.335
station_origin[Caledonian Road & Barnsbury]	-0.3721	0.044	-8.507	0.000	-0.458	-0.286
station_origin[Cambridge Heath]	-0.1308	0.056	-2.341	0.019	-0.240	-0.021
station_origin[Camden Road]	0.4536	0.030	15.106	0.000	0.395	0.512
station_origin[Camden Town]	-0.2356	0.019	-12.158	0.000	-0.274	-0.198
station_origin[Canada Water]	1.8160	0.011	171.799	0.000	1.795	1.837
station_origin[Canary Wharf]	1.2457	0.012	104.096	0.000	1.222	1.269
station_origin[Canning Town]	1.6142	0.011	145.100	0.000	1.592	1.636
station_origin[Cannon Street]	-0.1275	0.018	-7.022	0.000	-0.163	-0.092
station_origin[Canonbury]	1.0332	0.022	46.903	0.000	0.990	1.076
station_origin[Canons Park]	-0.0884	0.023	-3.882	0.000	-0.133	-0.044
station_origin[Carpenders Park]	1.2070	0.029	41.822	0.000	1.150	1.264
station_origin[Chadwell Heath]	2.6681	0.018	144.664	0.000	2.632	2.704
station_origin[Chalfont & Latimer]	-0.5473	0.034	-16.084	0.000	-0.614	-0.481
station_origin[Chalk Farm]	-0.4746	0.022	-21.329	0.000	-0.518	-0.431

station_origin[Chancery Lane]	-1.3572	0.031	-44.161	0.000	-1.417	-1.297
station_origin[Charing Cross]	0.2105	0.015	14.037	0.000	0.181	0.240
station_origin[Chesham]	-0.5119	0.034	-14.975	0.000	-0.579	-0.445
station_origin[Cheshunt]	0.1003	0.077	1.302	0.193	-0.051	0.251
station_origin[Chigwell]	-1.3755	0.047	-29.239	0.000	-1.468	-1.283
station_origin[Chingford]	2.1094	0.024	88.304	0.000	2.063	2.156
station_origin[Chiswick Park]	-0.7593	0.029	-25.888	0.000	-0.817	-0.702
station_origin[Chorleywood]	-0.6300	0.035	-18.155	0.000	-0.698	-0.562
station_origin[Clapham Common]	0.2460	0.017	14.653	0.000	0.213	0.279
station_origin[Clapham High Street]	1.4360	0.027	52.869	0.000	1.383	1.489
station_origin[Clapham Junction]	2.4342	0.015	163.988	0.000	2.405	2.463
station_origin[Clapham North]	-0.2898	0.021	-13.989	0.000	-0.330	-0.249
station_origin[Clapham South]	0.5902	0.015	38.943	0.000	0.560	0.620
station_origin[Clapton]	1.5294	0.027	56.074	0.000	1.476	1.583
station_origin[Cockfosters]	-0.8948	0.033	-27.142	0.000	-0.959	-0.830
station_origin[Colindale]	0.5961	0.016	36.331	0.000	0.564	0.628
station_origin[Colliers Wood]	0.5434	0.016	33.560	0.000	0.512	0.575
station_origin[Covent Garden]	-2.1935	0.049	-44.477	0.000	-2.290	-2.097
station_origin[Crossharbour]	0.6192	0.026	24.240	0.000	0.569	0.669
station_origin[Crouch Hill]	-0.0433	0.051	-0.843	0.399	-0.144	0.057
station_origin[Croxley]	-0.5418	0.033	-16.663	0.000	-0.606	-0.478
station_origin[Crystal Palace]	1.8507	0.024	75.963	0.000	1.803	1.898
station_origin[Custom House]	-0.0448	0.037	-1.218	0.223	-0.117	0.027
station_origin[Cutty Sark]	0.9829	0.023	42.035	0.000	0.937	1.029
station_origin[Cyprus]	-0.0361	0.046	-0.792	0.428	-0.125	0.053
station_origin[Dagenham East]	-0.2714	0.025	-11.021	0.000	-0.320	-0.223
station_origin[Dagenham Heathway]	0.2161	0.020	11.054	0.000	0.178	0.254
station_origin[Dalston Junction]	1.6873	0.018	93.147	0.000	1.652	1.723
station_origin[Dalston Kingsland]	1.4181	0.020	69.226	0.000	1.378	1.458
station_origin[Debden]	-0.3502	0.026	-13.239	0.000	-0.402	-0.298
station_origin[Denmark Hill]	1.6490	0.025	66.575	0.000	1.600	1.698
station_origin[Deptford Bridge]	1.2041	0.022	55.009	0.000	1.161	1.247
station_origin[Devons Road]	0.4058	0.028	14.393	0.000	0.351	0.461

station_origin[Dollis Hill]	-0.2429	0.022	-11.153	0.000	-0.286	-0.200
station_origin[Ealing Broadway]	1.3757	0.013	108.978	0.000	1.351	1.400
station_origin[Ealing Common]	-0.4286	0.024	-17.788	0.000	-0.476	-0.381
station_origin[Earl's Court]	0.6477	0.014	46.548	0.000	0.620	0.675
station_origin[East Acton]	-0.3058	0.022	-13.613	0.000	-0.350	-0.262
station_origin[East Finchley]	0.5641	0.016	35.496	0.000	0.533	0.595
station_origin[East Ham]	0.8343	0.014	57.922	0.000	0.806	0.862
station_origin[East India]	0.5758	0.025	22.756	0.000	0.526	0.625
station_origin[East Putney]	0.4607	0.016	28.386	0.000	0.429	0.493
station_origin[Eastcote]	0.0118	0.022	0.535	0.592	-0.031	0.055
station_origin[Edgware]	0.2246	0.019	11.536	0.000	0.186	0.263
station_origin[Edgware Road]	-0.4508	0.021	-21.833	0.000	-0.491	-0.410
station_origin[Edmonton Green]	1.6467	0.024	69.574	0.000	1.600	1.693
station_origin[Elephant & Castle]	0.6311	0.013	46.956	0.000	0.605	0.657
station_origin[Elm Park]	0.1435	0.021	6.742	0.000	0.102	0.185
station_origin[Elverson Road]	0.5331	0.031	17.318	0.000	0.473	0.593
station_origin[Embankment]	-0.2116	0.018	-11.889	0.000	-0.246	-0.177
station_origin[Emerson Park]	2.5747	0.055	46.701	0.000	2.467	2.683
station_origin[Enfield Town]	2.0219	0.025	80.294	0.000	1.973	2.071
station_origin[Epping]	0.2704	0.021	12.963	0.000	0.229	0.311
station_origin[Euston]	1.2307	0.011	108.724	0.000	1.209	1.253
station_origin[Euston Square]	0.2709	0.017	16.239	0.000	0.238	0.304
station_origin[Fairlop]	-0.8273	0.033	-25.021	0.000	-0.892	-0.762
station_origin[Farringdon]	0.1393	0.016	8.447	0.000	0.107	0.172
station_origin[Finchley Central]	0.6995	0.016	44.936	0.000	0.669	0.730
station_origin[Finchley Road]	0.2662	0.017	15.661	0.000	0.233	0.300
station_origin[Finchley Road & Frognal]	0.5252	0.031	17.025	0.000	0.465	0.586
station_origin[Finsbury Park]	1.7936	0.011	165.619	0.000	1.772	1.815
station_origin[Forest Gate]	1.8940	0.022	85.004	0.000	1.850	1.938
station_origin[Forest Hill]	1.8863	0.022	85.739	0.000	1.843	1.929
station_origin[Fulham Broadway]	-0.1399	0.020	-7.161	0.000	-0.178	-0.102
station_origin[Gallions Reach]	0.1825	0.041	4.492	0.000	0.103	0.262
station_origin[Gants Hill]	0.5549	0.017	33.490	0.000	0.522	0.587

station_origin[Gidea Park]	2.6358	0.019	135.280	0.000	2.598	2.674
station_origin[Gloucester Road]	0.0780	0.017	4.627	0.000	0.045	0.111
station_origin[Golders Green]	0.3884	0.017	22.813	0.000	0.355	0.422
station_origin[Goldhawk Road]	-0.7297	0.029	-25.448	0.000	-0.786	-0.673
station_origin[Goodge Street]	-2.3651	0.053	-44.650	0.000	-2.469	-2.261
station_origin[Goodmayes]	2.4054	0.020	119.642	0.000	2.366	2.445
station_origin[Gospel Oak]	0.3224	0.032	10.226	0.000	0.261	0.384
station_origin[Grange Hill]	-1.1440	0.040	-28.437	0.000	-1.223	-1.065
station_origin[Great Portland Street]	-1.0747	0.031	-34.756	0.000	-1.135	-1.014
station_origin[Green Park]	-0.8880	0.024	-37.674	0.000	-0.934	-0.842
station_origin[Greenford]	-0.0157	0.021	-0.735	0.462	-0.057	0.026
station_origin[Greenwich]	0.6218	0.028	21.920	0.000	0.566	0.677
station_origin[Gunnersbury]	-0.2888	0.023	-12.652	0.000	-0.334	-0.244
station_origin[Hackney Central]	1.5042	0.019	78.186	0.000	1.467	1.542
station_origin[Hackney Downs]	0.6014	0.035	17.119	0.000	0.533	0.670
station_origin[Hackney Wick]	0.6860	0.027	25.051	0.000	0.632	0.740
station_origin[Haggerston]	0.9761	0.024	41.263	0.000	0.930	1.022
station_origin[Hainault]	0.1795	0.020	8.891	0.000	0.140	0.219
station_origin[Hammersmith]	1.1056	0.013	87.013	0.000	1.081	1.130
station_origin[Hampstead]	-0.5518	0.025	-22.483	0.000	-0.600	-0.504
station_origin[Hampstead Heath]	0.0734	0.034	2.148	0.032	0.006	0.140
station_origin[Hanger Lane]	-0.3421	0.024	-14.148	0.000	-0.389	-0.295
station_origin[Harlesden]	-0.7228	0.027	-26.429	0.000	-0.776	-0.669
station_origin[Harold Wood]	2.5720	0.021	124.995	0.000	2.532	2.612
station_origin[Harringay Green Lanes]	0.5629	0.045	12.463	0.000	0.474	0.651
station_origin[Harrow & Wealdstone]	-0.1905	0.026	-7.407	0.000	-0.241	-0.140
station_origin[Harrow-on-the-Hill]	0.8045	0.016	51.727	0.000	0.774	0.835
station_origin[Hatch End]	1.2104	0.035	34.212	0.000	1.141	1.280
station_origin[Hatton Cross]	-0.5263	0.029	-18.326	0.000	-0.583	-0.470
station_origin[Headstone Lane]	0.3292	0.047	7.002	0.000	0.237	0.421
station_origin[Heathrow Terminal 4]	-0.9471	0.036	-26.048	0.000	-1.018	-0.876
station_origin[Heathrow Terminal 5]	-0.9097	0.034	-26.766	0.000	-0.976	-0.843
station_origin[Heathrow Terminals 2 & 3]	-0.1878	0.023	-8.067	0.000	-0.233	-0.142

station_origin[Hendon Central]	0.2693	0.018	14.848	0.000	0.234	0.305
station_origin[Heron Quays]	0.5060	0.025	20.179	0.000	0.457	0.555
station_origin[High Barnet]	0.2112	0.020	10.666	0.000	0.172	0.250
station_origin[High Street Kensington]	-0.6159	0.023	-26.522	0.000	-0.661	-0.570
station_origin[Highams Park]	2.4095	0.020	117.979	0.000	2.369	2.450
station_origin[Highbury & Islington]	1.7054	0.011	157.842	0.000	1.684	1.727
station_origin[Highgate]	0.2854	0.017	16.532	0.000	0.252	0.319
station_origin[Hillingdon]	-0.4819	0.029	-16.663	0.000	-0.539	-0.425
station_origin[Holborn]	-1.0921	0.026	-42.080	0.000	-1.143	-1.041
station_origin[Holland Park]	-0.8148	0.027	-30.243	0.000	-0.868	-0.762
station_origin[Holloway Road]	-0.2713	0.020	-13.484	0.000	-0.311	-0.232
station_origin[Homerton]	1.2857	0.019	67.677	0.000	1.248	1.323
station_origin[Honor Oak Park]	1.5094	0.025	59.520	0.000	1.460	1.559
station_origin[Hornchurch]	-0.2305	0.026	-8.876	0.000	-0.281	-0.180
station_origin[Hounslow Central]	-0.1972	0.023	-8.510	0.000	-0.243	-0.152
station_origin[Hounslow East]	-0.2280	0.023	-9.765	0.000	-0.274	-0.182
station_origin[Hounslow West]	-0.2356	0.024	-9.883	0.000	-0.282	-0.189
station_origin[Hoxton]	0.3080	0.032	9.590	0.000	0.245	0.371
station_origin[Hyde Park Corner]	-2.5360	0.063	-40.420	0.000	-2.659	-2.413
station_origin[Ickenham]	-0.8762	0.038	-23.042	0.000	-0.951	-0.802
station_origin[Ilford]	2.8763	0.017	174.285	0.000	2.844	2.909
station_origin[Imperial Wharf]	0.5255	0.035	15.001	0.000	0.457	0.594
station_origin[Island Gardens]	0.5596	0.028	20.000	0.000	0.505	0.614
station_origin[Kennington]	-0.2536	0.019	-13.487	0.000	-0.290	-0.217
station_origin[Kensal Green]	-0.5297	0.025	-21.245	0.000	-0.579	-0.481
station_origin[Kensal Rise]	0.9907	0.024	40.874	0.000	0.943	1.038
station_origin[Kensington]	-0.1556	0.049	-3.186	0.001	-0.251	-0.060
station_origin[Kentish Town]	-0.4616	0.022	-20.822	0.000	-0.505	-0.418
station_origin[Kentish Town West]	0.0754	0.038	1.963	0.050	9.82e-05	0.151
station_origin[Kenton]	-0.1067	0.032	-3.316	0.001	-0.170	-0.044
station_origin[Kew Gardens]	-0.2329	0.023	-10.129	0.000	-0.278	-0.188
station_origin[Kilburn]	0.2773	0.017	16.141	0.000	0.244	0.311
station_origin[Kilburn High Road]	0.2533	0.054	4.651	0.000	0.147	0.360

station_origin[Kilburn Park]	-0.5949	0.025	-24.097	0.000	-0.643	-0.547
station_origin[King George V]	0.3125	0.033	9.376	0.000	0.247	0.378
station_origin[King's Cross St. Pancras]	1.6922	0.010	163.528	0.000	1.672	1.712
station_origin[Kingsbury]	0.0013	0.021	0.061	0.951	-0.040	0.043
station_origin[Knightsbridge]	-1.3200	0.031	-42.520	0.000	-1.381	-1.259
station_origin[Ladbroke Grove]	-0.3743	0.022	-16.735	0.000	-0.418	-0.330
station_origin[Lambeth North]	-1.4124	0.031	-45.320	0.000	-1.473	-1.351
station_origin[Lancaster Gate]	-0.0905	0.019	-4.768	0.000	-0.128	-0.053
station_origin[Langdon Park]	0.4387	0.027	15.972	0.000	0.385	0.493
station_origin[Latimer Road]	-1.2719	0.035	-36.106	0.000	-1.341	-1.203
station_origin[Leicester Square]	-1.6035	0.033	-48.641	0.000	-1.668	-1.539
station_origin[Lewisham]	2.4281	0.014	169.225	0.000	2.400	2.456
station_origin[Leyton]	0.7698	0.015	53.036	0.000	0.741	0.798
station_origin[Leyton Midland Road]	1.3319	0.035	37.752	0.000	1.263	1.401
station_origin[Leytonstone]	0.6953	0.015	45.764	0.000	0.665	0.725
station_origin[Leytonstone High Road]	1.3322	0.041	32.625	0.000	1.252	1.412
station_origin[Limehouse]	1.3857	0.016	89.199	0.000	1.355	1.416
station_origin[Liverpool Street]	1.7764	0.010	175.130	0.000	1.757	1.796
station_origin[London Bridge]	1.5063	0.009	159.550	0.000	1.488	1.525
station_origin[London City Airport]	0.9751	0.024	40.803	0.000	0.928	1.022
station_origin[London Fields]	1.2225	0.031	38.863	0.000	1.161	1.284
station_origin[Loughton]	0.1003	0.021	4.709	0.000	0.059	0.142
station_origin[Maida Vale]	-0.5215	0.023	-22.341	0.000	-0.567	-0.476
station_origin[Manor House]	0.3482	0.016	21.534	0.000	0.317	0.380
station_origin[Manor Park]	1.6642	0.025	65.334	0.000	1.614	1.714
station_origin[Mansion House]	-1.7128	0.045	-38.191	0.000	-1.801	-1.625
station_origin[Marble Arch]	-0.8578	0.025	-34.245	0.000	-0.907	-0.809
station_origin[Maryland]	0.6925	0.035	19.860	0.000	0.624	0.761
station_origin[Marylebone]	0.2663	0.015	17.300	0.000	0.236	0.297
station_origin[Mile End]	0.5463	0.015	37.132	0.000	0.517	0.575
station_origin[Mill Hill East]	-0.7086	0.029	-24.182	0.000	-0.766	-0.651
station_origin[Moor Park]	-0.9274	0.040	-22.936	0.000	-1.007	-0.848
station_origin[Moorgate]	-0.0750	0.017	-4.341	0.000	-0.109	-0.041

station_origin[Morden]	0.9638	0.014	66.822	0.000	0.936	0.992
station_origin[Mornington Crescent]	-1.7056	0.039	-43.715	0.000	-1.782	-1.629
station_origin[Mudchute]	0.4595	0.028	16.126	0.000	0.404	0.515
station_origin[Neasden]	-0.2872	0.023	-12.711	0.000	-0.331	-0.243
station_origin[New Cross]	1.4869	0.028	53.778	0.000	1.433	1.541
station_origin[New Cross Gate]	1.3111	0.026	51.138	0.000	1.261	1.361
station_origin[Newbury Park]	0.4025	0.018	22.413	0.000	0.367	0.438
station_origin[North Acton]	-0.0323	0.020	-1.602	0.109	-0.072	0.007
station_origin[North Ealing]	-1.4139	0.045	-31.125	0.000	-1.503	-1.325
station_origin[North Greenwich]	1.0382	0.013	79.580	0.000	1.013	1.064
station_origin[North Harrow]	-0.2132	0.025	-8.637	0.000	-0.262	-0.165
station_origin[North Wembley]	-0.7994	0.032	-25.222	0.000	-0.862	-0.737
station_origin[Northfields]	0.0008	0.021	0.040	0.968	-0.039	0.041
station_origin[Northolt]	0.2881	0.019	15.089	0.000	0.251	0.326
station_origin[Northwick Park]	-0.2255	0.024	-9.553	0.000	-0.272	-0.179
station_origin[Northwood]	-0.0996	0.024	-4.129	0.000	-0.147	-0.052
station_origin[Northwood Hills]	-0.3670	0.028	-13.175	0.000	-0.422	-0.312
station_origin[Norwood Junction]	0.9101	0.039	23.087	0.000	0.833	0.987
station_origin[Notting Hill Gate]	0.1754	0.017	10.525	0.000	0.143	0.208
station_origin[Oakwood]	-0.3797	0.026	-14.890	0.000	-0.430	-0.330
station_origin[Old Street]	0.1097	0.016	6.805	0.000	0.078	0.141
station_origin[Osterley]	-0.6315	0.028	-22.486	0.000	-0.687	-0.576
station_origin[Oval]	-0.1827	0.019	-9.739	0.000	-0.219	-0.146
station_origin[Oxford Circus]	-0.6032	0.021	-29.076	0.000	-0.644	-0.563
station_origin[Paddington]	1.3449	0.010	132.677	0.000	1.325	1.365
station_origin[Park Royal]	-1.0552	0.043	-24.396	0.000	-1.140	-0.970
station_origin[Parsons Green]	0.1723	0.018	9.824	0.000	0.138	0.207
station_origin[Peckham Rye]	1.6364	0.022	74.102	0.000	1.593	1.680
station_origin[Penge West]	0.2800	0.053	5.257	0.000	0.176	0.384
station_origin[Perivale]	-0.4951	0.026	-18.711	0.000	-0.547	-0.443
station_origin[Piccadilly Circus]	-1.5874	0.032	-49.328	0.000	-1.650	-1.524
station_origin[Pimlico]	-0.1394	0.018	-7.672	0.000	-0.175	-0.104
station_origin[Pinner]	0.2045	0.021	9.802	0.000	0.164	0.245

station_origin[Plaistow]	-0.0004	0.019	-0.023	0.982	-0.037	0.036
station_origin[Pontoon Dock]	0.8941	0.024	37.355	0.000	0.847	0.941
station_origin[Poplar]	0.6898	0.023	30.563	0.000	0.646	0.734
station_origin[Preston Road]	-0.1860	0.023	-8.257	0.000	-0.230	-0.142
station_origin[Prince Regent]	0.3533	0.034	10.452	0.000	0.287	0.420
station_origin[Pudding Mill Lane]	-0.6102	0.049	-12.479	0.000	-0.706	-0.514
station_origin[Putney Bridge]	-0.1725	0.020	-8.436	0.000	-0.213	-0.132
station_origin[Queen's Park]	0.3687	0.016	22.463	0.000	0.337	0.401
station_origin[Queens Road Peckham]	1.3804	0.024	56.879	0.000	1.333	1.428
station_origin[Queensbury]	0.0807	0.021	3.894	0.000	0.040	0.121
station_origin[Queensway]	-0.3121	0.021	-14.791	0.000	-0.353	-0.271
station_origin[Ravenscourt Park]	-0.6914	0.027	-25.723	0.000	-0.744	-0.639
station_origin[Rayners Lane]	0.2472	0.019	12.776	0.000	0.209	0.285
station_origin[Rectory Road]	1.2646	0.034	37.731	0.000	1.199	1.330
station_origin[Redbridge]	-0.3335	0.024	-13.980	0.000	-0.380	-0.287
station_origin[Regent's Park]	-2.1242	0.053	-40.359	0.000	-2.227	-2.021
station_origin[Richmond]	0.6751	0.017	40.509	0.000	0.642	0.708
station_origin[Rickmansworth]	-0.4304	0.030	-14.460	0.000	-0.489	-0.372
station_origin[Roding Valley]	-1.4455	0.050	-28.649	0.000	-1.544	-1.347
station_origin[Romford]	2.8911	0.018	163.538	0.000	2.856	2.926
station_origin[Rotherhithe]	0.9400	0.026	35.712	0.000	0.888	0.992
station_origin[Royal Albert]	0.0247	0.040	0.614	0.539	-0.054	0.103
station_origin[Royal Oak]	-0.8462	0.028	-30.159	0.000	-0.901	-0.791
station_origin[Royal Victoria]	1.0416	0.024	44.088	0.000	0.995	1.088
station_origin[Ruislip]	-0.4774	0.029	-16.339	0.000	-0.535	-0.420
station_origin[Ruislip Gardens]	-0.6693	0.034	-19.953	0.000	-0.735	-0.604
station_origin[Ruislip Manor]	-0.4690	0.028	-16.562	0.000	-0.525	-0.414
station_origin[Russell Square]	-1.1235	0.027	-41.420	0.000	-1.177	-1.070
station_origin[Seven Kings]	2.4296	0.020	123.646	0.000	2.391	2.468
station_origin[Seven Sisters]	1.6554	0.011	144.258	0.000	1.633	1.678
station_origin[Shadwell]	1.1153	0.014	78.745	0.000	1.088	1.143
station_origin[Shenfield]	0.7412	0.064	11.549	0.000	0.615	0.867
station_origin[Shepherd's Bush]	0.9719	0.013	74.222	0.000	0.946	0.998

station_origin[Shepherd's Bush Market]	-0.4939	0.025	-19.590	0.000	-0.543	-0.444
station_origin[Shoreditch High Street]	0.5669	0.034	16.780	0.000	0.501	0.633
station_origin[Silver Street]	1.4622	0.026	55.473	0.000	1.411	1.514
station_origin[Sloane Square]	0.1079	0.017	6.491	0.000	0.075	0.140
station_origin[Snaresbrook]	-0.4715	0.025	-18.555	0.000	-0.521	-0.422
station_origin[South Acton]	0.2548	0.040	6.316	0.000	0.176	0.334
station_origin[South Ealing]	-0.3028	0.023	-13.004	0.000	-0.348	-0.257
station_origin[South Hampstead]	-0.0249	0.064	-0.389	0.697	-0.150	0.101
station_origin[South Harrow]	-0.3936	0.027	-14.415	0.000	-0.447	-0.340
station_origin[South Kensington]	0.1036	0.017	6.266	0.000	0.071	0.136
station_origin[South Kenton]	-0.6952	0.032	-21.796	0.000	-0.758	-0.633
station_origin[South Quay]	0.5387	0.026	20.844	0.000	0.488	0.589
station_origin[South Ruislip]	-0.4414	0.028	-15.574	0.000	-0.497	-0.386
station_origin[South Tottenham]	0.5280	0.049	10.671	0.000	0.431	0.625
station_origin[South Wimbledon]	0.2905	0.019	15.567	0.000	0.254	0.327
station_origin[South Woodford]	0.3388	0.018	18.782	0.000	0.303	0.374
station_origin[Southbury]	0.6050	0.044	13.843	0.000	0.519	0.691
station_origin[Southfields]	0.4552	0.017	27.561	0.000	0.423	0.488
station_origin[Southgate]	0.1953	0.019	10.333	0.000	0.158	0.232
station_origin[Southwark]	0.0288	0.017	1.728	0.084	-0.004	0.061
station_origin[St James Street]	1.4331	0.032	44.680	0.000	1.370	1.496
station_origin[St. James's Park]	-0.8919	0.025	-35.626	0.000	-0.941	-0.843
station_origin[St. John's Wood]	-0.2080	0.020	-10.595	0.000	-0.246	-0.170
station_origin[St. Paul's]	-1.4340	0.035	-41.170	0.000	-1.502	-1.366
station_origin[Stamford Brook]	-0.5238	0.025	-20.974	0.000	-0.573	-0.475
station_origin[Stamford Hill]	0.5472	0.046	11.918	0.000	0.457	0.637
station_origin[Stanmore]	0.0611	0.022	2.834	0.005	0.019	0.103
station_origin[Star Lane]	-0.5055	0.049	-10.419	0.000	-0.601	-0.410
station_origin[Stepney Green]	-0.3922	0.021	-18.647	0.000	-0.433	-0.351
station_origin[Stockwell]	0.9043	0.013	69.431	0.000	0.879	0.930
station_origin[Stoke Newington]	1.0235	0.031	32.515	0.000	0.962	1.085
station_origin[Stonebridge Park]	-0.7366	0.028	-26.778	0.000	-0.791	-0.683
station_origin[Stratford]	2.6001	0.010	270.721	0.000	2.581	2.619

station_origin[Stratford High Street]	-0.4841	0.052	-9.309	0.000	-0.586	-0.382
station_origin[Stratford International]	1.1552	0.023	51.169	0.000	1.111	1.199
station_origin[Sudbury Hill]	-0.6244	0.029	-21.621	0.000	-0.681	-0.568
station_origin[Sudbury Town]	-0.4487	0.027	-16.324	0.000	-0.503	-0.395
station_origin[Surrey Quays]	1.6816	0.019	89.079	0.000	1.645	1.719
station_origin[Swiss Cottage]	-0.0699	0.019	-3.682	0.000	-0.107	-0.033
station_origin[Sydenham]	1.6301	0.025	65.415	0.000	1.581	1.679
station_origin[Temple]	-2.2918	0.054	-42.266	0.000	-2.398	-2.185
station_origin[Theobalds Grove]	1.1228	0.045	25.142	0.000	1.035	1.210
station_origin[Theydon Bois]	-0.9837	0.040	-24.838	0.000	-1.061	-0.906
station_origin[Tooting Bec]	0.6633	0.015	43.839	0.000	0.634	0.693
station_origin[Tooting Broadway]	1.0218	0.014	74.993	0.000	0.995	1.049
station_origin[Tottenham Court Road]	-1.0616	0.025	-42.113	0.000	-1.111	-1.012
station_origin[Tottenham Hale]	0.9737	0.014	70.161	0.000	0.946	1.001
station_origin[Totteridge & Whetstone]	-0.1226	0.023	-5.376	0.000	-0.167	-0.078
station_origin[Tower Gateway]	0.3511	0.035	9.946	0.000	0.282	0.420
station_origin[Tower Hill]	0.5713	0.014	41.392	0.000	0.544	0.598
station_origin[Tufnell Park]	-0.2987	0.021	-14.195	0.000	-0.340	-0.257
station_origin[Turkey Street]	0.9918	0.037	27.161	0.000	0.920	1.063
station_origin[Turnham Green]	0.1204	0.019	6.405	0.000	0.084	0.157
station_origin[Turnpike Lane]	0.6170	0.015	40.743	0.000	0.587	0.647
station_origin[Upminster]	-0.5433	0.038	-14.205	0.000	-0.618	-0.468
station_origin[Upminster Bridge]	-0.8815	0.037	-23.795	0.000	-0.954	-0.809
station_origin[Upney]	-0.4604	0.026	-17.938	0.000	-0.511	-0.410
station_origin[Upper Holloway]	-0.0878	0.053	-1.671	0.095	-0.191	0.015
station_origin[Upton Park]	0.5242	0.016	33.304	0.000	0.493	0.555
station_origin[Uxbridge]	0.1759	0.021	8.290	0.000	0.134	0.218
station_origin[Vauxhall]	1.3127	0.012	112.104	0.000	1.290	1.336
station_origin[Victoria]	1.9966	0.010	200.871	0.000	1.977	2.016
station_origin[Walthamstow Central]	1.6774	0.012	142.824	0.000	1.654	1.700
station_origin[Walthamstow Queens Road]	1.5675	0.033	48.026	0.000	1.504	1.631
station_origin[Wandsworth Road]	0.5498	0.041	13.370	0.000	0.469	0.630
station_origin[Wanstead]	-0.3604	0.024	-14.868	0.000	-0.408	-0.313

station_origin[Wanstead Park]	1.4694	0.037	39.613	0.000	1.397	1.542
station_origin[Wapping]	0.7708	0.029	26.222	0.000	0.713	0.828
station_origin[Warren Street]	-0.8682	0.024	-35.581	0.000	-0.916	-0.820
station_origin[Warwick Avenue]	-0.4443	0.022	-19.943	0.000	-0.488	-0.401
station_origin[Waterloo]	2.2621	0.009	262.929	0.000	2.245	2.279
station_origin[Watford]	-0.3171	0.028	-11.173	0.000	-0.373	-0.262
station_origin[Watford High Street]	0.6532	0.047	13.882	0.000	0.561	0.745
station_origin[Watford Junction]	0.7476	0.046	16.254	0.000	0.657	0.838
station_origin[Wembley Central]	-0.1855	0.023	-8.064	0.000	-0.231	-0.140
station_origin[Wembley Park]	0.8162	0.015	54.558	0.000	0.787	0.845
station_origin[West Acton]	-0.8029	0.031	-26.241	0.000	-0.863	-0.743
station_origin[West Brompton]	-0.0506	0.018	-2.754	0.006	-0.087	-0.015
station_origin[West Croydon]	1.6482	0.029	57.223	0.000	1.592	1.705
station_origin[West Finchley]	-0.5236	0.027	-19.380	0.000	-0.577	-0.471
station_origin[West Ham]	0.8292	0.014	60.132	0.000	0.802	0.856
station_origin[West Hampstead]	0.9302	0.013	70.697	0.000	0.904	0.956
station_origin[West Harrow]	-0.8558	0.033	-26.229	0.000	-0.920	-0.792
station_origin[West India Quay]	-1.4819	0.079	-18.754	0.000	-1.637	-1.327
station_origin[West Kensington]	-0.3197	0.021	-15.163	0.000	-0.361	-0.278
station_origin[West Ruislip]	-0.5613	0.032	-17.629	0.000	-0.624	-0.499
station_origin[West Silvertown]	0.0782	0.034	2.268	0.023	0.011	0.146
station_origin[Westbourne Park]	-0.3436	0.022	-15.489	0.000	-0.387	-0.300
station_origin[Westferry]	0.7219	0.023	31.528	0.000	0.677	0.767
station_origin[Westminster]	-1.1113	0.027	-41.092	0.000	-1.164	-1.058
station_origin[White City]	-0.3054	0.022	-13.827	0.000	-0.349	-0.262
station_origin[White Hart Lane]	0.9841	0.032	30.403	0.000	0.921	1.048
station_origin[Whitechapel]	1.0286	0.012	82.585	0.000	1.004	1.053
station_origin[Willesden Green]	0.4912	0.016	30.696	0.000	0.460	0.523
station_origin[Willesden Junction]	0.4625	0.016	28.166	0.000	0.430	0.495
station_origin[Wimbledon]	1.0761	0.015	71.739	0.000	1.047	1.105
station_origin[Wimbledon Park]	-0.4308	0.026	-16.470	0.000	-0.482	-0.380
station_origin[Wood Green]	0.6337	0.015	41.736	0.000	0.604	0.664
station_origin[Wood Lane]	-0.3966	0.032	-12.573	0.000	-0.458	-0.335

```
station_origin[Wood Street]                          1.6639      0.028     58.557      0.000
      1.608        1.720
station_origin[Woodford]                             0.5108      0.017     30.013      0.000
      0.477        0.544
station_origin[Woodgrange Park]                      1.3528      0.044     30.635      0.000
      1.266        1.439
station_origin[Woodside Park]                        0.0803      0.020      3.976      0.000
      0.041        0.120
station_origin[Woolwich Arsenal]                     2.5141      0.014    174.224      0.000
      2.486        2.542
log_jobs                                             0.7378      0.001   1154.912      0.000
      0.737        0.739
log_dist                                            -0.3672      0.001   -572.567      0.000
     -0.368       -0.366
==========================================================================================
=======================
```

The γ parameter related to the destination attractiveness: 0.7378

The β distance decay parameter: -0.3672.

P value shows all the explainatory variables are statistically significant <0.01. The z score indicates that the jobs have the most influence on the model.

In [21]:

```
#create some Oi and Dj columns in the dataframe and store row and column totals in them:
#to create O_i, take cdatasub ...then... group by origcodenew ...then... summarise by ca
O_i = pd.DataFrame(df_drop.groupby(["station_origin"])["flows"].agg(np.sum))
O_i.rename(columns={"flows":"O_i"}, inplace = True)
df_drop = df_drop.merge(O_i, on = "station_origin", how = "left" )

D_j = pd.DataFrame(df_drop.groupby(["station_destination"])["flows"].agg(np.sum))
D_j.rename(columns={"flows":"D_j"}, inplace = True)
df_drop = df_drop.merge(D_j, on = "station_destination", how = "left" )
```

```
C:\Users\lenovo\AppData\Local\Temp\ipykernel_15608\2847122089.py:3: FutureWarning: The p
rovided callable <function sum at 0x000002137F7B09A0> is currently using SeriesGroupBy.s
um. In a future version of pandas, the provided callable will be used directly. To keep
current behavior pass the string "sum" instead.
  O_i = pd.DataFrame(df_drop.groupby(["station_origin"])["flows"].agg(np.sum))
C:\Users\lenovo\AppData\Local\Temp\ipykernel_15608\2847122089.py:7: FutureWarning: The p
rovided callable <function sum at 0x000002137F7B09A0> is currently using SeriesGroupBy.s
um. In a future version of pandas, the provided callable will be used directly. To keep
current behavior pass the string "sum" instead.
  D_j = pd.DataFrame(df_drop.groupby(["station_destination"])["flows"].agg(np.sum))
```

In [22]:

```
coefs = pd.DataFrame(prodSim.params)
coefs.reset_index(inplace=True)
coefs.rename(columns = {0:"alpha_i", "index":"coef"}, inplace = True)
to_repl = ["(station_origin)", "\[", "\]"]
for x in to_repl:
    coefs["coef"] = coefs["coef"].str.replace(x, "",regex=True)
coefs
df_drop = df_drop.merge(coefs, left_on="station_origin", right_on="coef", how = "left")
df_drop.drop(columns = ["coef"], inplace = True)
df_drop.head()
```

Out[22]:

	station_origin	station_destination	flows	population	jobs	distance	flow	log_flow	dist	log_
0	Abbey Road	Beckton	1	599	442	8510.121774	1	0.000000	8511.121775	9.04
1	Abbey Road	Blackwall	3	599	665	3775.448872	3	1.098612	3776.448873	8.23
2	Abbey Road	Canary Wharf	1	599	58772	5086.514220	1	0.000000	5087.514221	8.53
3	Abbey Road	Canning Town	37	599	15428	2228.923167	37	3.610918	2229.923168	7.70

4	Abbey Road	Crossharbour	1	599	1208	6686.475560	1	0.000000	6687.475561	8.80

In [24]:
```
alpha_i = prodSim.params[0:-2]
gamma = prodSim.params[-2]
beta = -prodSim.params[-1]
```

```
C:\Users\lenovo\AppData\Local\Temp\ipykernel_15608\1887813951.py:2: FutureWarning: Serie
s.__getitem__ treating keys as positions is deprecated. In a future version, integer key
s will always be treated as labels (consistent with DataFrame behavior). To access a val
ue by position, use `ser.iloc[pos]`
  gamma = prodSim.params[-2]
C:\Users\lenovo\AppData\Local\Temp\ipykernel_15608\1887813951.py:3: FutureWarning: Serie
s.__getitem__ treating keys as positions is deprecated. In a future version, integer key
s will always be treated as labels (consistent with DataFrame behavior). To access a val
ue by position, use `ser.iloc[pos]`
  beta = -prodSim.params[-1]
```

In [25]:
```
df_drop["prodsimest1"] = np.exp(df_drop["alpha_i"]+gamma*df_drop["log_jobs"]
                                 - beta*df_drop["log_dist"])
#or you could do it the easy way like we did last week with the fitted column (See previ
df_drop.head(10)
```

Out[25]:

	station_origin	station_destination	flows	population	jobs	distance	flow	log_flow	dist	log_
0	Abbey Road	Beckton	1	599	442	8510.121774	1	0.000000	8511.121775	9.04
1	Abbey Road	Blackwall	3	599	665	3775.448872	3	1.098612	3776.448873	8.23
2	Abbey Road	Canary Wharf	1	599	58772	5086.514220	1	0.000000	5087.514221	8.53
3	Abbey Road	Canning Town	37	599	15428	2228.923167	37	3.610918	2229.923168	7.70
4	Abbey Road	Crossharbour	1	599	1208	6686.475560	1	0.000000	6687.475561	8.80
5	Abbey Road	Cutty Sark	2	599	1748	8503.898909	2	0.693147	8504.898910	9.04
6	Abbey Road	Cyprus	7	599	850	6532.099618	7	1.945910	6533.099619	8.78
7	Abbey Road	Devons Road	1	599	611	3958.324171	1	0.000000	3959.324172	8.28
8	Abbey Road	East India	2	599	1522	3384.141666	2	0.693147	3385.141667	8.12
9	Abbey Road	Island Gardens	2	599	691	7706.296370	2	0.693147	7707.296371	8.94

Assessing the model output

In [ ]:
```

```

To test the "Goodness-of-fit" of the model, check the coefficient of determination (r^2) or the Square Root of Mean Squared Error (RMSE).

In [45]:
```
import scipy.stats

def CalcRSqaured(observed, estimated):
    """Calculate the r^2 from a series of observed and estimated target values
    inputs:
    Observed: Series of actual observed values
    estimated: Series of predicted values"""

    r, p = scipy.stats.pearsonr(observed, estimated)
    R2 = r **2

    return R2
```

In [46]:
```
def CalcRMSE(observed, estimated):
```

```
    """Calculate Root Mean Square Error between a series of observed and estimated value
    inputs:
    Observed: Series of actual observed values
    estimated: Series of predicted values"""

    res = (observed -estimated)**2
    RMSE = round(sqrt(res.mean()), 3)

    return RMSE
```

Let's use poisson regression to test which model can get the best result:

Flow - Distance:

In [47]:
```
best2_formula_double_sim_exp = "flow ~ station_origin + log_jobs + dist-1"

best2_double_sim_exp = smf.glm(formula=best2_formula_double_sim_exp, data=df_origin, fam

print("R-squared: ", CalcRSqaured(df_origin["flow"], best2_double_sim_exp.mu))
print("RMSE: ", CalcRMSE(df_origin["flow"], best2_double_sim_exp.mu))
```

```
R-squared:  0.45590488914083815
RMSE:  114.37
```

Flow - Logged distance:

In [48]:
```
formula_double_sim_exp = "flow ~ station_origin + log_jobs + log_dist-1"

double_sim_exp = smf.glm(formula=formula_double_sim_exp, data=df_origin, family=sm.famil

print("R-squared: ", CalcRSqaured(df_origin["flow"], double_sim_exp.mu))
print("RMSE: ", CalcRMSE(df_origin["flow"], double_sim_exp.mu))
```

```
R-squared:  0.1683779372123963
RMSE:  152.287
```

Logged flow - distance:

In [49]:
```
best_formula_double_sim_exp = "log_flow ~ station_origin + log_jobs + dist-1"

best_double_sim_exp = smf.glm(formula=best_formula_double_sim_exp, data=df_origin, famil

# print
print("R-squared: ", CalcRSqaured(df_origin["log_flow"], best_double_sim_exp.mu))
print("RMSE: ", CalcRMSE(df_origin["log_flow"], best_double_sim_exp.mu))
```

```
R-squared:  0.5387247573433747
RMSE:  1.088
```

The r square is the biggest and the RMSE is the smallest, therefore this combination has the best performance.

Logged flow - logged distance:

In [50]:
```
formula_double_sim_exp = "log_flow ~ station_origin + log_jobs + log_dist-1"

double_sim_exp = smf.glm(formula=formula_double_sim_exp, data=df_origin, family=sm.famil

# print
print("R-squared: ", CalcRSqaured(df_origin["log_flow"], double_sim_exp.mu))
print("RMSE: ", CalcRMSE(df_origin["log_flow"], double_sim_exp.mu))
```

```
R-squared:  0.30449397993579636
```

```
RMSE:  1.44
```

IV. Scenarios

IV.1. Scenario A: Job Decrease

Cut half of the jobs in Canary Wharf while conserving the number of people commuting:

IV.2. Scenario B: Travel Cost Increase

Increase in travel cost can lead to decrease in the population parameter: I don't know how to calculate it.

IV.3. Discussion

Reference

Berche, B., C. von Ferber, T. Holovatch, and Yu. Holovatch. 2009. “Resilience of Public Transport Networks against Attacks.” The European Physical Journal B 71 (1): 125–37. https://doi.org/10.1140/epjb/e2009-00291-3.

Bloch, Francis, Matthew O. Jackson, and Pietro Tebaldi. 2019. “Centrality Measures in Networks.” SSRN Scholarly Paper ID 2749124. Rochester, NY: Social Science Research Network. https://doi.org/10.2139/ssrn.2749124

Bonacich, Phillip. 1987. “Power and Centrality: A Family of Measures.” American Journal of Sociology 92 (5): 1170–82. https://doi.org/10.1086/228631.

Dijkstra, E.W. (1959). A note on two problems in connexion with graphs. Numer. Math. 1, 269–271 . https://doi.org/10.1007/BF01386390

Freeman, L. (1977): “A set of measures of centrality based on betweenness,” Sociometry, 40(1), 35–41.

Latora, Vito, and Massimo Marchiori. “Efficient behavior of small-world networks.” Physical Review Letters 87.19 (2001): 198701. https://doi.org/10.1103/PhysRevLett.87.198701

Nguyen, Quang & Khanh, Nguyen & Cassi, Davide & Bellingeri, Michele. (2021). New Betweenness Centrality Node Attack Strategies for Real-World Complex Weighted Networks. Complexity. 2021. https://doi.org/10.1155/2021/1677445.

Wilson, A. G. (1971). ‘A Family of Spatial Interaction Models, and Associated Developments’. Environment and Planning A: Economy and Space. SAGE Publications Ltd, 3 (1), pp. 1–32. doi: 10.1068/a030001.