

# CSC-8101 ENGINEERING FOR AI

## Neo4j COURSE-WORK

NAME: KAILASH BALACHANDIRAN  
STUDENT ID: 220243160

### SCENARIO:

A taxi company operating in New York City (NYC) is rethinking its fleet allocation strategy. Their restructuring process is motivated by recent changes in passenger travel patterns (provoked by Covid) and the arrival of new competitors. To minimise costs, the company wants to station its fleet in a maximum of 20 city zones, spread across the city. At the same time, the company wants to maximise trips served and therefore choose zones that have hub-like characteristics, i.e. that are well-connected and near centres of high user activity. Lastly, the company considers operating purely outside the Manhattan borough where there is less competition.

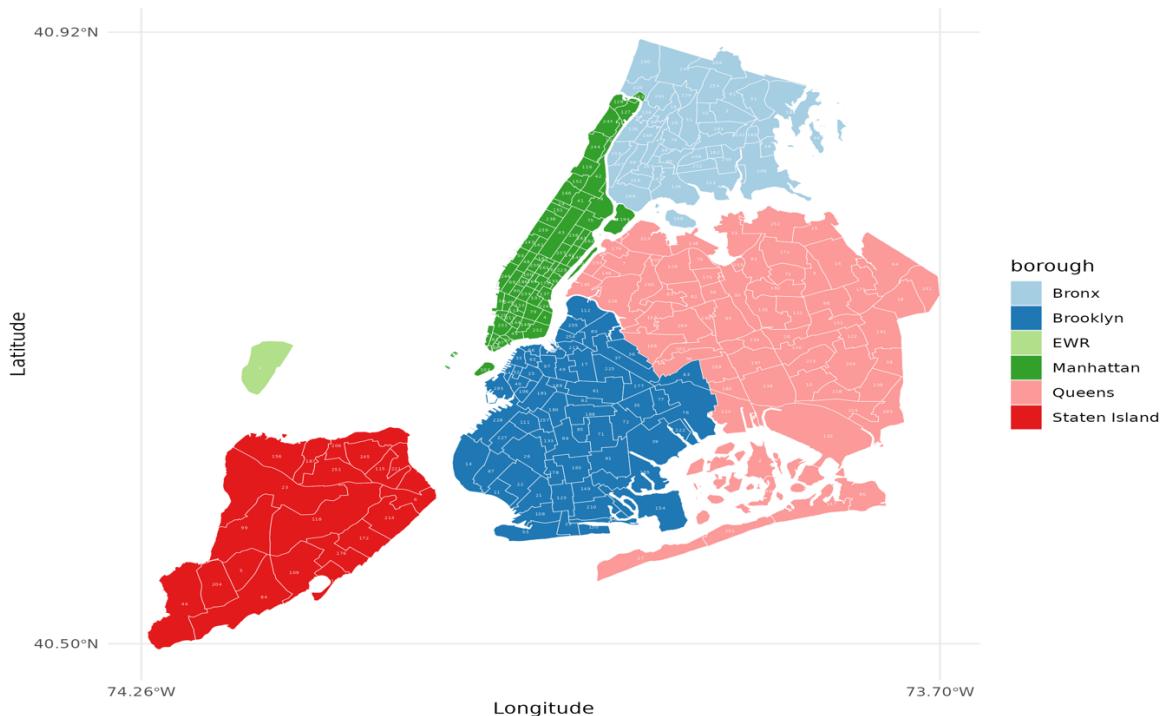


Figure 1.1

To help with this task, the taxi company contracts you to analyse its historical trip records and recommend a list of city zones where they should station their fleets. To that end, the company gives you a copy of its Neo4j graph database, available here, containing aggregate statistics of taxi trips undertaken in 2021. There are two types of nodes: borough and zone. Two zones are connected by a relationship of type :CONNECTS if a minimum of one trip was recorded between them. In addition, CONNECTS relationships have property trips representing the total number of trips observed in the year. Lastly, a node is related to a borough via a relationship of type: IN.

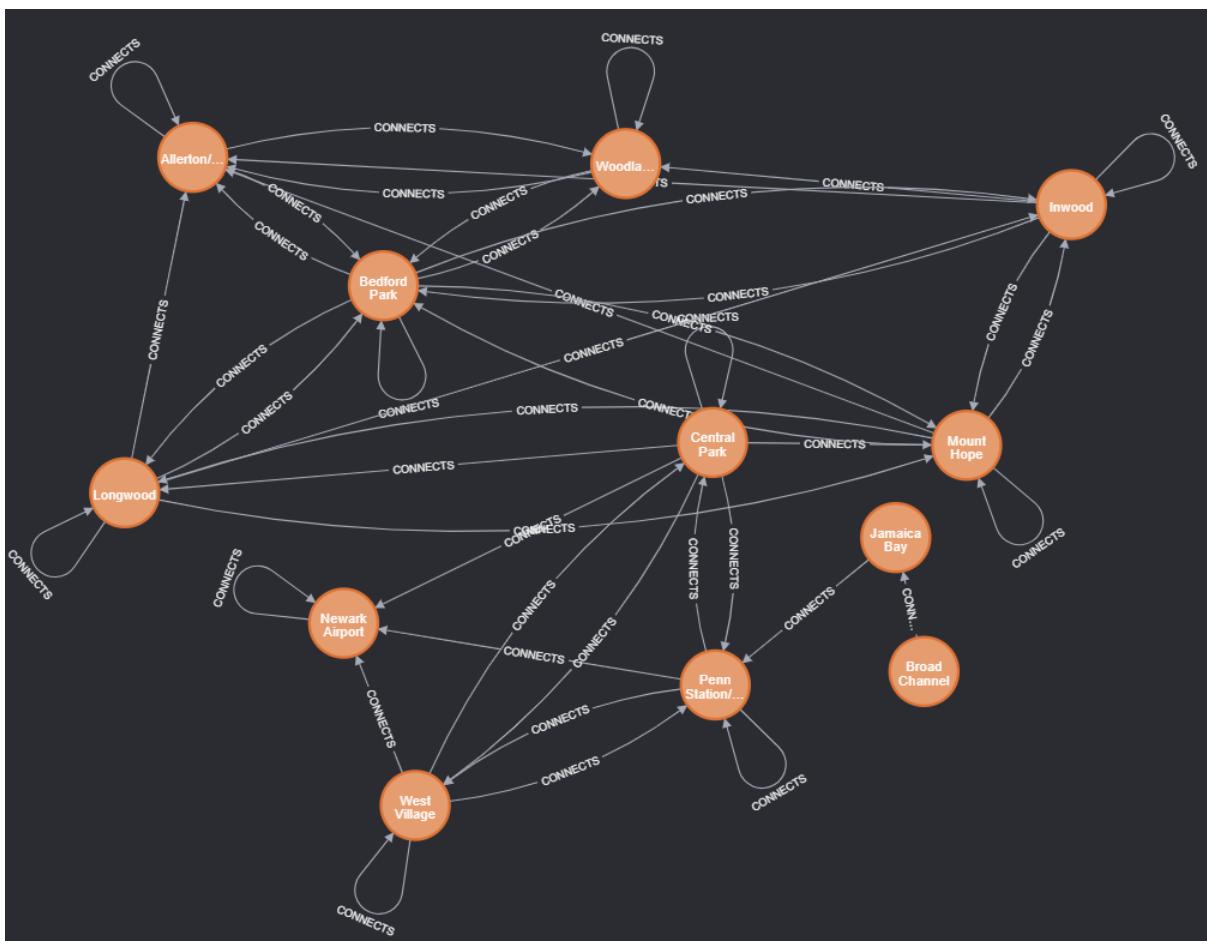


Figure 1.2

Based on preliminary exploration of the database and the NYC zones and boroughs map (shown above), you decide to tackle the challenge by combining two methods of network analysis available in the Neo4j Data Science Library. Your action plan is divided into three parts:

1. Perform community detection to identify clusters of strongly connected zones.
2. Perform centrality analysis to measure the hub-like features of each zone.

3. Combine the results above by identifying the top 3 zones with highest centrality score within each community cluster. This process is done twice for the entire city:
  - i. once including Manhattan, and
  - ii. once excluding Manhattan.

In addition to your Neo4j analysis, you will use the visualisation app below to help interpret the results of your network analysis and present them to your client (the taxi company). To use the app, export your results at the end of each stage to a csv file and load it into the app (more details given in each task).

- <http://csc8101-neo4j-shiny.eksouth.cloudapp.azure.com/>

## TASK DETAILS:

There are four tasks:

1. Find isolated nodes
2. Compute the community cluster of each node
3. Compute the centrality score of each node
4. Find the top centrality zones within each community

In task 0 you will develop a cypher query to find nodes and relationship of a certain type.

In tasks 1 and 2 you will execute two different algorithms available in the Neo4j Data Science Library (GDS). Both tasks follow the typical workflow described here:

- Create an (in-memory) graph projection.
- Estimate the memory necessary to run the algorithm (optional).
- Run the algorithm in stats mode to summarise the output of the algorithm.
- Run the algorithm in stream mode to run the algorithm but not store the results in the original graph.

In task 3 you will develop a cypher query that combines the outcomes of tasks 1 and 2.

Due to database writing restrictions, the coursework is split across three neo4j databases - one for Task 0, another for tasks 1 and 2 and a third for task 3.

Go here for connection details.

## TASK 0 - Find isolated nodes

Find all:

1. Self-pointing relationships, that is, all relationship instances of type: CONNECTS where the start node is the same as the end node.

```
MATCH (k)-[r:CONNECTS]->(k)
RETURN k,r
```



Figure 0.1

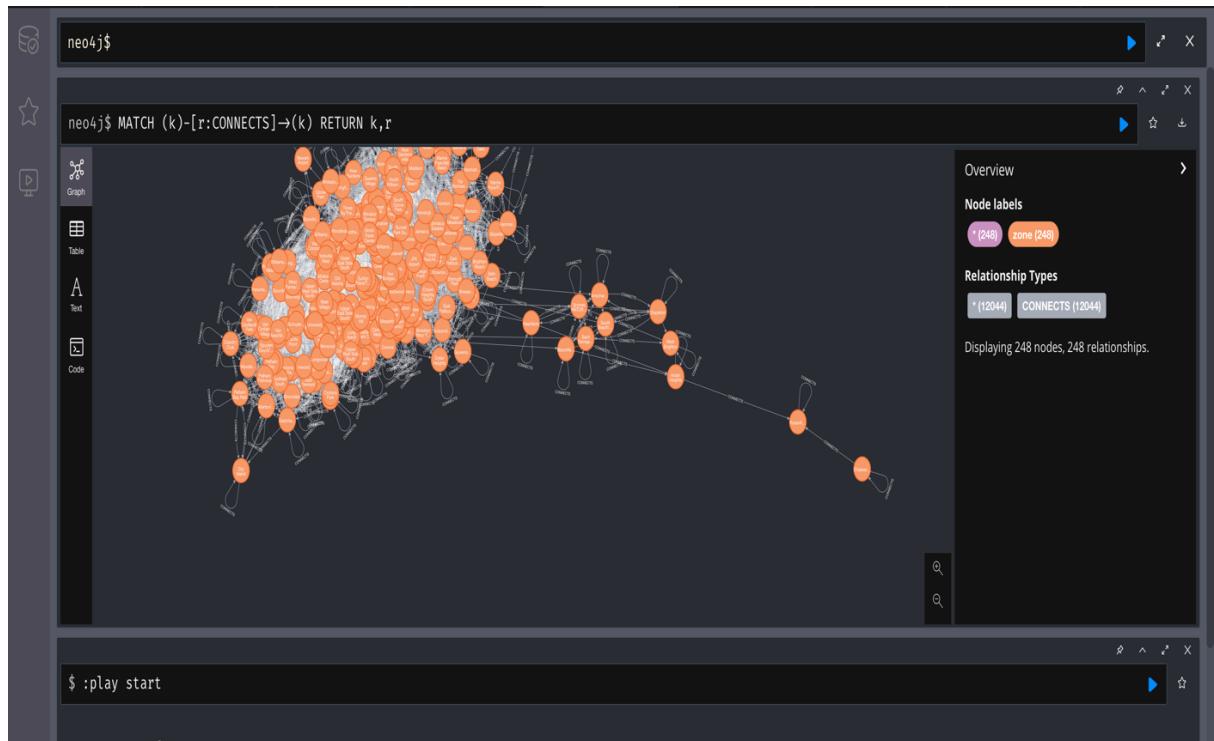


Figure 0.2

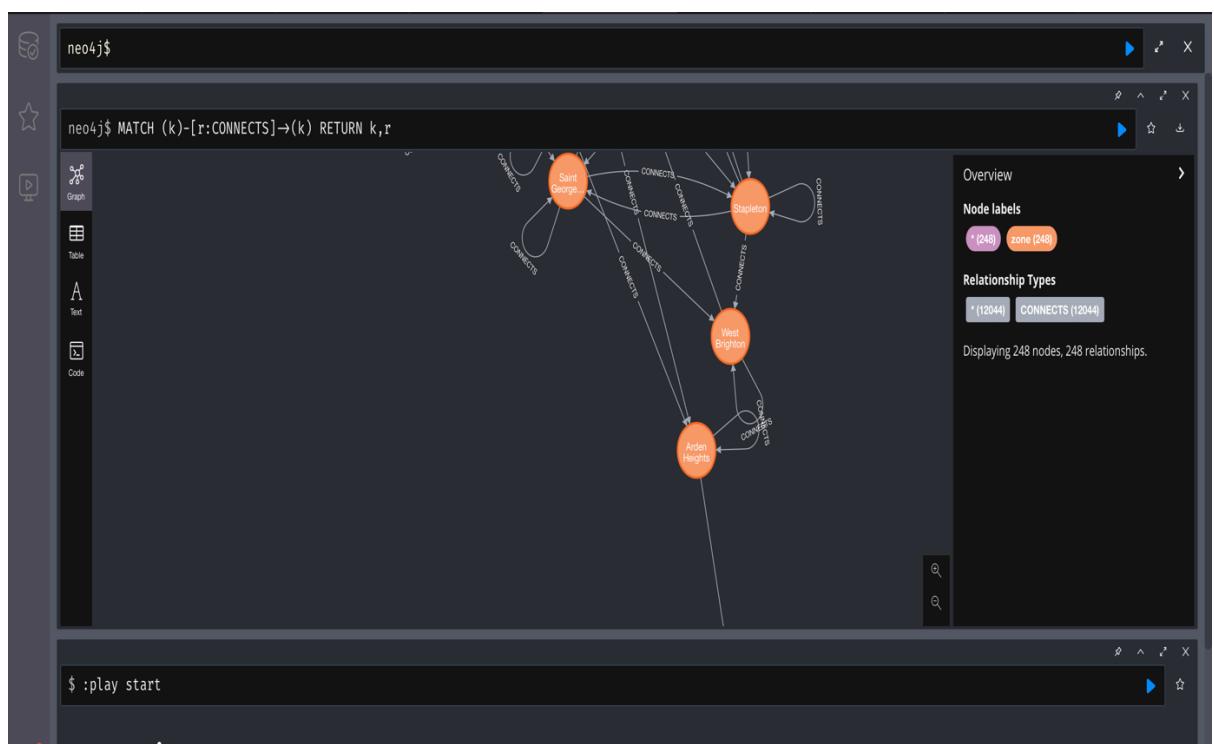
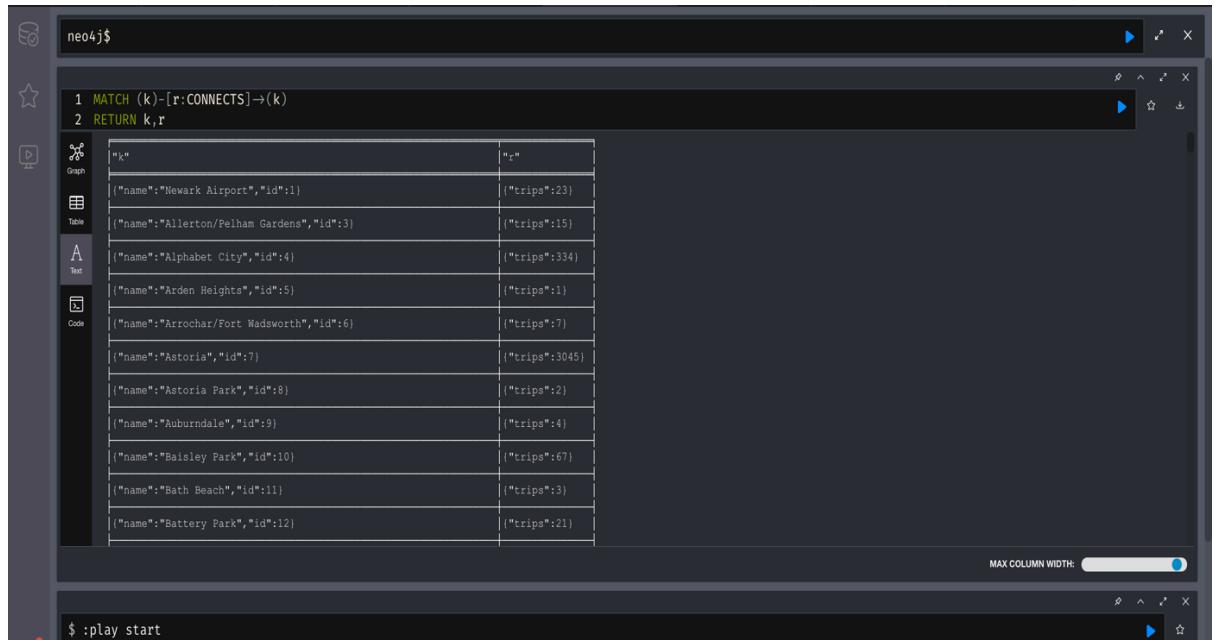


Figure 0.3



The screenshot shows the Neo4j browser interface with a query results table. The query is:

```

1 MATCH (k)-[r:CONNECTS]->(k)
2 RETURN k,r
  
```

The results table contains 12 rows of data:

k	r
{"name": "Newark Airport", "id": 1}	{"trips": 23}
{"name": "Allerton/Pelham Gardens", "id": 3}	{"trips": 15}
{"name": "Alphabet City", "id": 4}	{"trips": 334}
{"name": "Arden Heights", "id": 5}	{"trips": 1}
{"name": "Arrrochar/Fort Wadsworth", "id": 6}	{"trips": 7}
{"name": "Astoria", "id": 7}	{"trips": 3045}
{"name": "Astoria Park", "id": 8}	{"trips": 2}
{"name": "Auburndale", "id": 9}	{"trips": 4}
{"name": "Baisley Park", "id": 10}	{"trips": 67}
{"name": "Bath Beach", "id": 11}	{"trips": 3}
{"name": "Battery Park", "id": 12}	{"trips": 21}

At the bottom of the interface, there is a command line with the text:

```
$ :play start
```

Figure 0.4

2. Find all isolated nodes, i.e., all nodes that have no relationship instances of type: CONNECTS to other nodes except possibly to themselves.

```

MATCH (k)
WHERE NOT (k)-[:CONNECTS]-() AND NOT (k)--(k)
RETURN k
  
```



Figure 0.5

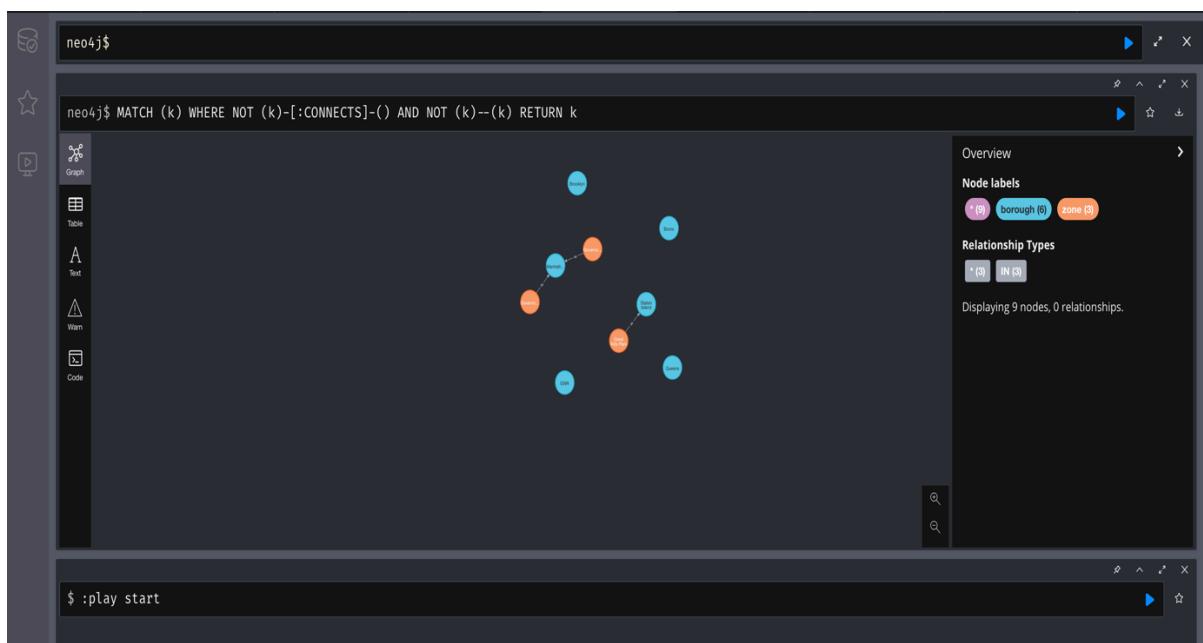
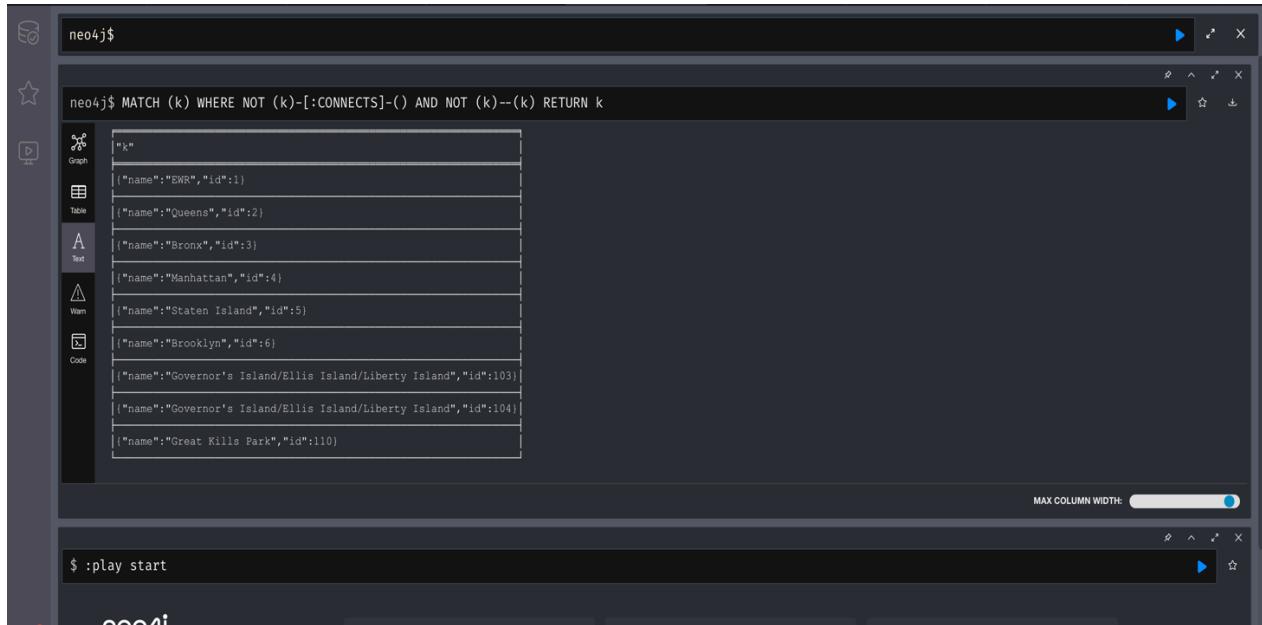


Figure 0.6



The screenshot shows the Neo4j browser interface. The top bar has the title 'neo4j\$'. Below it is a search bar with the query: 'MATCH (k) WHERE NOT (k)-[:CONNECTS]-() AND NOT (k)--(k) RETURN k'. The main area displays a table with the following data:

k
{"name": "EWR", "id": 1}
{"name": "Queens", "id": 2}
{"name": "Bronx", "id": 3}
{"name": "Manhattan", "id": 4}
{"name": "Staten Island", "id": 5}
{"name": "Brooklyn", "id": 6}
{"name": "Governor's Island/Ellis Island/Liberty Island", "id": 103}
{"name": "Governor's Island/Ellis Island/Liberty Island", "id": 104}
{"name": "Great Kills Park", "id": 110}

At the bottom of the interface, there is a command line with '\$ :play start' and a play button.

Figure 0.7

### Task 1 - Community detection

Run the Louvain algorithm for community detection, available in the GDS library, with the following arguments:

- Graph projection of type UNDIRECTED.
- Name the graph projection as USERNAME-communities where USERNAME is your student number (this is necessary to avoid naming conflicts between students).
- Weighted by the trips property in :CONNECTS type of relationships.

```
CALL gds.graph.create(
  '220243160-communities_7892',
  'zone',
  {
    CONNECTS: {
      type: 'CONNECTS',
      orientation: 'UNDIRECTED',
      properties: 'trips'
    }
  }
)
```

```
CALL gds.louvain.stats('220243160-communities_7892')
YIELD communityCount
```

```
CALL gds.louvain.stream('220243160-communities_7892', {
  relationshipWeightProperty: 'trips' })
YIELD nodeId, communityId
RETURN gds.util.asNode(nodeId).id AS zone_id,communityId AS community_id
```

nodeProjection	relationshipProjection	graphName	nodeCount	relationshipCount	createMillis
<pre>{   "zone": {     "label": "zone",     "properties": {       "trips": {         "defaultValue": null,         "property": "trips",         "aggregation": "DEFAULT"       }     }   } }</pre>		'220243160-communities_7892'	259	23716	19

Started streaming 1 records in less than 1 ms and completed after 20 ms.

\$ :play start

Figure 1.1

nodeProjection	relationshipProjection	graphName	nodeCount	relationshipCount	createMillis
<pre>{"zone":{"label":"zone","properties":{}} {CONNECTS:{orientation:"UNDIRECTED",aggregation:"DEFAULT",type:"CONNECTS",properties:{trips:{defaultValue:null,property:"trips",aggregation:"DEFAULT"}}}}</pre>		'220243160-communities_7892'	259	23716	19

\$ :play start

Figure 1.2

As specified above, run the algorithm in 2 modes: stats and stream:

- Report the number of communities using the stats mode,
- In stream mode, return the id and community properties of each zone (in this order),
- Export the results of running the algorithm in stream as a CSV file with two columns named zone\_id, community\_id and
- Visualise the results in the app by uploading the produced csv file (right-click to download image).

The screenshot shows the Neo4j Browser interface. The top bar has the title "neo4j\$". The main area contains a code editor with the following query:

```
1 CALL gds.louvain.stats('220243160-communities_7892')
2 YIELD communityCount
```

Below the code editor is a "Table" panel showing the results:

communityCount
5

The bottom bar has the command "\$ :play start".

Figure 1.3

The screenshot shows the Neo4j Browser interface. The top bar has the title "neo4j\$". The main area contains a code editor with the same query as Figure 1.3:

```
1 CALL gds.louvain.stats('220243160-communities_7892')
2 YIELD communityCount
```

Below the code editor is a "Table" panel showing the results:

communityCount
5

Below the table, a message states "Started streaming 1 records in less than 1 ms and completed after 659 ms." The bottom bar has the command "\$ :play start".

Figure 1.4

The screenshot shows the neo4j browser interface. In the top right, there are three buttons: a blue play button, a green refresh button, and a red X button. Below these are two smaller buttons: a grey star and a grey square with a double arrow. On the left side, there's a sidebar with icons for a database, a star, a play button, and a refresh symbol. The main area has a dark background with white text. At the top, it says "neo4j\$". Below that is a code block with the following query:

```
1 CALL gds.louvain.stream('220243160-communities_7892', { relationshipWeightProperty: 'trips' })
2 YIELD nodeId, communityId
3 RETURN gds.util.asNode(nodeId).id AS zone_id,communityId AS community_id
```

Below the code, there's a table with two columns: "zone\_id" and "community\_id". The data is as follows:

zone_id	community_id
3	234
4	95
5	216
6	216
7	127
8	127
9	127
10	127
11	95

At the bottom of the table, it says "Started streaming 259 records after 1 ms and completed after 205 ms."

Figure 1.5

This screenshot is similar to Figure 1.5, showing the neo4j browser with the same query and table structure. However, the "MAX COLUMN WIDTH:" slider at the bottom right is set to a higher value, which causes the "community\_id" column to be displayed as a single long string of numbers separated by vertical bars instead of individual cells. The data appears as follows:

zone_id	community_id
3	234
4	95
5	216
6	216
7	127
8	127
9	127
10	127
11	95

Figure 1.6

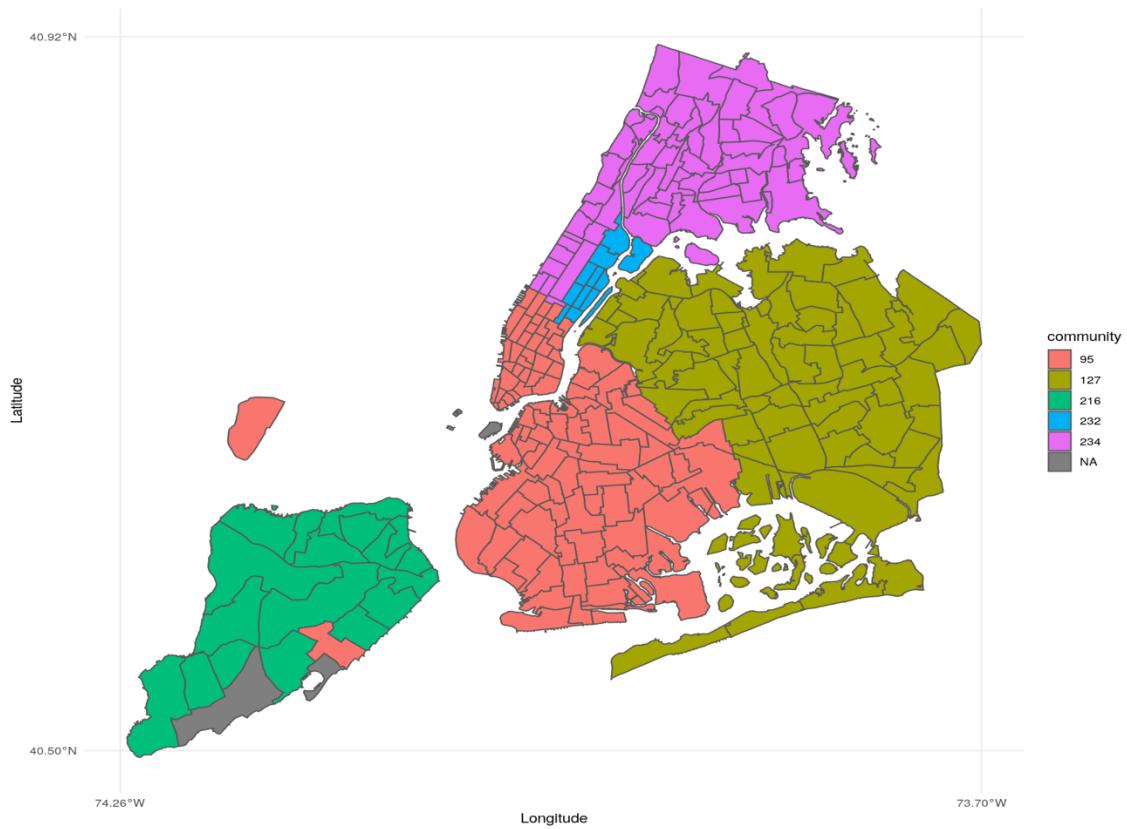


Figure 1.7

## Task 2 - Centrality analysis

Run the Page Rank algorithm for centrality analysis, available in the GDS library, with the following arguments:

- Directed graph projection
- Name the graph projection as USERNAME-centrality where USERNAME is your student number (this is necessary to avoid naming conflicts between students).
- Damping factor: 0.75
- Weighted by the trips property in :CONNECTS type of relationships.

As specified above, run the algorithm in 2 modes: stats and stream:

- Report the maximum and minimum centrality score using the stats mode,
- In stream mode, return the id, centrality properties of each zone (in this order),
- Export the results of running the algorithm in stream as a CSV file with two columns named zone\_id and centrality\_score and
- Visualise the results in the app by uploading the produced csv file (right-click to download image).

```
CALL gds.graph.create(  
  '220243160-centrality_789',  
  'zone',  
  'CONNECTS',  
  
  {  
    relationshipProperties: 'trips'  
  }  
)
```

```
CALL gds.pageRank.write.estimate('220243160-centrality_789', {  
  writeProperty: 'pageRank',  
  maxIterations: 20,  
  dampingFactor: 0.75  
})  
YIELD nodeCount, relationshipCount, bytesMin, bytesMax, requiredMemory
```

```
CALL gds.pageRank.stats('220243160-centrality_789', {  
  dampingFactor: 0.75,  
  relationshipWeightProperty: 'trips'  
})  
YIELD centralityDistribution  
RETURN centralityDistribution.max AS MAX , centralityDistribution.min AS MIN
```

```
CALL gds.pageRank.stream('220243160-centrality_789', {  
  dampingFactor: 0.75,  
  relationshipWeightProperty: 'trips'  
})  
YIELD nodeId, score  
RETURN gds.util.asNode(nodeId).id AS zone_id, score AS centrality_score  
ORDER BY score DESC
```

```

neo4j$ CALL gds.graph.create(
  '220243160-centrality_789',
  'zone',
  'CONNECTS',
  {
    relationshipProperties: 'trips'
  }
)
  
```

nodeProjection	relationshipProjection	graphName	nodeCount	relationshipCount	createMillis
<pre>         {           "zone": {             "label": "zone",             "properties": {}           }         }       </pre>	<pre>         {           "CONNECTS": {             "orientation": "NATURAL",             "aggregation": "DEFAULT",             "type": "CONNECTS",             "properties": {               "trips": {                 "defaultValue": null,                 "property": "trips",                 "aggregation": "DEFAULT"               }             }           }         }       </pre>	"220243160-centrality_789"	259	11858	12

Started streaming 1 records in less than 1 ms and completed after 13 ms.

Figure 2.1

```

neo4j$ CALL gds.graph.create(
  '220243160-centrality_789',
  'zone',
  'CONNECTS',
  {
    relationshipProperties: 'trips'
  }
)
  
```

nodeProjection	relationshipProjection	graphName	nodeCount	relationshipCount	createMillis
<pre>         {           "zone": {             "label": "zone",             "properties": {}           }         }       </pre>	<pre>         {           "CONNECTS": {             "orientation": "NATURAL",             "aggregation": "DEFAULT",             "type": "CONNECTS",             "properties": {               "trips": {                 "defaultValue": null,                 "property": "trips",                 "aggregation": "DEFAULT"               }             }           }         }       </pre>	"220243160-centrality_789"	259	11858	12

Started streaming 1 records in less than 1 ms and completed after 13 ms.

Figure 2.2

```

neo4j$ CALL gds.pageRank.write.estimate('220243160-centrality_789', {
  writeProperty: 'pageRank',
  maxIterations: 20,
  dampingFactor: 0.75
})
  
```

nodeCount	relationshipCount	bytesMin	bytesMax	requiredMemory
259	11858	7040	7040	"7040 Bytes"

MAX COLUMN WIDTH:

Figure 2.3

The screenshot shows the neo4j browser interface with a query window titled "neo4j\$". The query is:

```
1 CALL gds.pageRank.write.estimate('220243160-centrality_789', {  
2   writeProperty: 'pageRank',  
3   maxIterations: 20,  
4   dampingFactor: 0.75  
5 })  
6 YIELD nodeCount, relationshipCount, bytesMin, bytesMax, requiredMemory
```

Below the query, a table displays the results:

	nodeCount	relationshipCount	bytesMin	bytesMax	requiredMemory
1	259	11858	7040	7040	"7040 Bytes"

At the bottom of the browser window, a message states: "Started streaming 1 records in less than 1 ms and completed after 1 ms."

Figure 2.4

The screenshot shows the neo4j browser interface with a query window titled "neo4j\$". The query is:

```
1 CALL gds.pageRank.stats('220243160-centrality_789', {  
2   dampingFactor: 0.75,  
3   relationshipWeightProperty: 'trips'  
4 })  
5 YIELD centralityDistribution  
6 RETURN centralityDistribution.max AS MAX , centralityDistribution.min AS MIN
```

Below the query, a table displays the results:

	MAX	MIN
1	3.660612106323242	0.25

At the bottom of the browser window, a message states: "Started streaming 1 records in less than 1 ms and completed after 51 ms."

Figure 2.5

The screenshot shows the neo4j browser interface with a query window titled "neo4j\$". The query is:

```
1 CALL gds.pageRank.stats('220243160-centrality_789', {  
2   dampingFactor: 0.75,  
3   relationshipWeightProperty: 'trips'  
4 })  
5 YIELD centralityDistribution  
6 RETURN centralityDistribution.max AS MAX , centralityDistribution.min AS MIN
```

The results are displayed in a table:

"MAX"	"MIN"
3.660612106323242	0.25

Below the table are three tabs: "Table", "Text", and "Code". A status bar at the bottom right shows "MAX COLUMN WIDTH: 100".

Figure 2.6

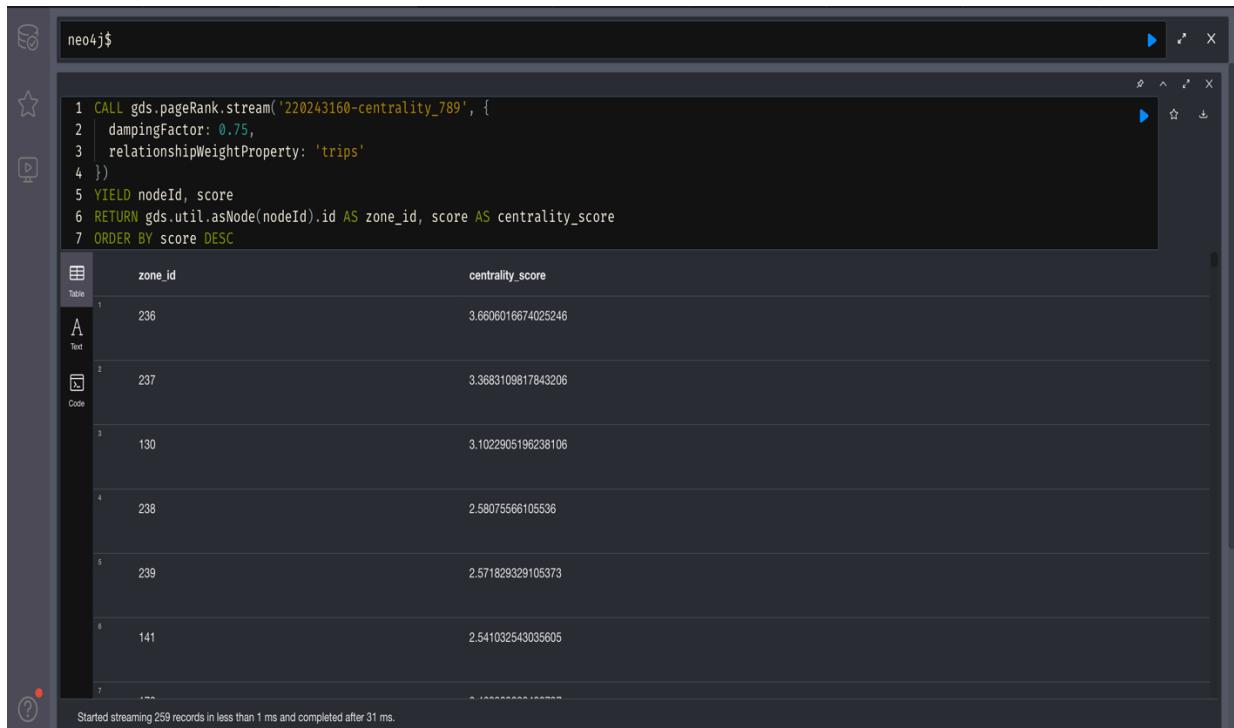
The screenshot shows the neo4j browser interface with a query window titled "neo4j\$". The query is:

```
1 CALL gds.pageRank.stream('220243160-centrality_789', {  
2   dampingFactor: 0.75,  
3   relationshipWeightProperty: 'trips'  
4 })  
5 YIELD nodeId, score  
6 RETURN gds.util.asNode(nodeId).id AS zone_id, score AS centrality_score  
7 ORDER BY score DESC
```

The results are displayed in a table:

"zone_id"	"centrality_score"
236	3.6606016674025246
237	3.3683109817843206
130	3.1022905196238106
238	2.58075566105536
239	2.571829329105373
141	2.541032543035605
170	2.462983399400797
161	2.450765535972166
74	2.450277911024033
75	2.356714805574518
61	2.319703481625731

Figure 2.7



The screenshot shows the Neo4j browser interface. In the top-left, there's a terminal window titled "neo4j\$". Below it is a code editor with a query:

```

1 CALL gds.pageRank.stream('220243160-centrality_789', {
2   dampingFactor: 0.75,
3   relationshipWeightProperty: 'trips'
4 })
5 YIELD nodeId, score
6 RETURN gds.util.asNode(nodeId).id AS zone_id, score AS centrality_score
7 ORDER BY score DESC
  
```

On the right, a table displays the results:

	zone_id	centrality_score
1	236	3.6606016674025246
2	237	3.3683109817843206
3	130	3.1022905196238106
4	238	2.58075566105536
5	239	2.571829329105373
6	141	2.541032543035605
7		

At the bottom left, a note says "Started streaming 259 records in less than 1 ms and completed after 31 ms."

Figure 2.8

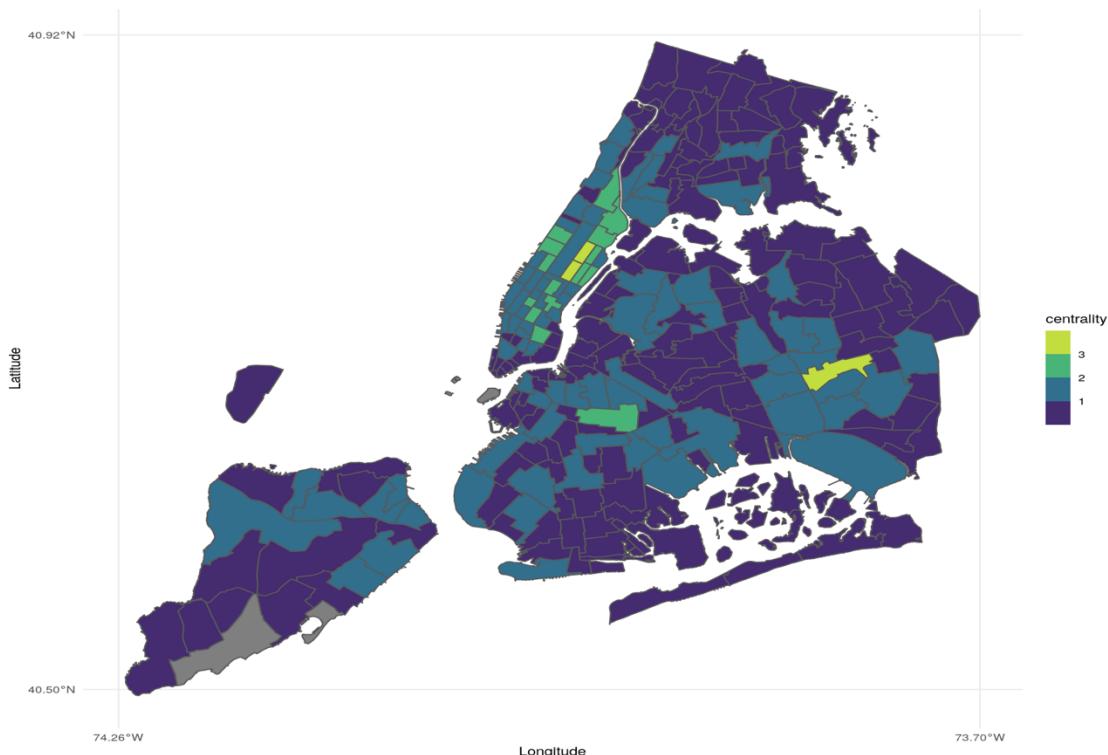


Figure 2.9

### Task 3 - Zone recommendation

Find the top 3 highest centrality zones per each community:

1. Including zones in the 'Manhattan' borough.
2. Excluding zones in the 'Manhattan' borough.

Do this using the available zone properties community (representing the community id obtained in 1) and centrality (representing the centrality score obtained in 2).

For each case, return two columns: zone\_id and community\_id and export the query results to a csv file. Then, using the visualisationapp, upload these (one at a time) to the 'Task-3' file input and produce a separate and produce two separate images.

```
MATCH (n:zone)
with n order by n.centrality desc
with n.community as class,collect({id:n.id, score:n.centrality}) as listt
UNWIND listt[0..3] AS l
return l.id as zone_id ,class as community_id
order by class
```

```
MATCH (n:zone)-[r:IN]->(b:borough) WHERE b.name <> 'Manhattan' with n order by
n.centrality desc
with n.community as class,collect({id:n.id, score:n.centrality}) as listt
UNWIND listt[0..3] AS l
return l.id as zone_id ,class as community_id
order by class
```

The screenshot shows the Neo4j browser interface with a query results window. The query is:neo4j\$ MATCH (n:zone) with n order by n.centrality desc with n.community as class,collect({id:n.id, score:n.centrality}) as listt UNWIND listt[...]The results are displayed as a table with two columns: "zone\_id" and "community\_id". The data is as follows:

zone_id	community_id
170	78
161	78
79	78
61	95
181	95
251	95
130	127
82	127
95	127
236	232
237	232

Figure 3.1

The screenshot shows the Neo4j browser interface with a query results window. The query is identical to Figure 3.1:neo4j\$ MATCH (n:zone) with n order by n.centrality desc with n.community as class,collect({id:n.id, score:n.centrality}) as listt UNWIND listt[...]The results are displayed as a table with two columns: "zone\_id" and "community\_id". The data is as follows:

zone_id	community_id
1	170
2	161
3	79
4	61
5	181
6	251
7	130

Started streaming 15 records in less than 1 ms and completed after 1 ms.

Figure 3.2

The screenshot shows the Neo4j browser interface with a query results table. The table has two columns: 'zone\_id' and 'community\_id'. The data is as follows:

zone_id	community_id
1	78
2	78
3	78
4	95
5	95
6	95
7	95

Started streaming 13 records in less than 1 ms and completed after 1 ms.

Figure 3.3

The screenshot shows the Neo4j browser interface with a query results table. The table has two columns: 'zone\_id' and 'community\_id'. The data is as follows:

zone_id	community_id
256	78
255	78
112	78
61	95
181	95
251	95
130	127
82	127
95	127
168	232
213	234

Figure 3.4

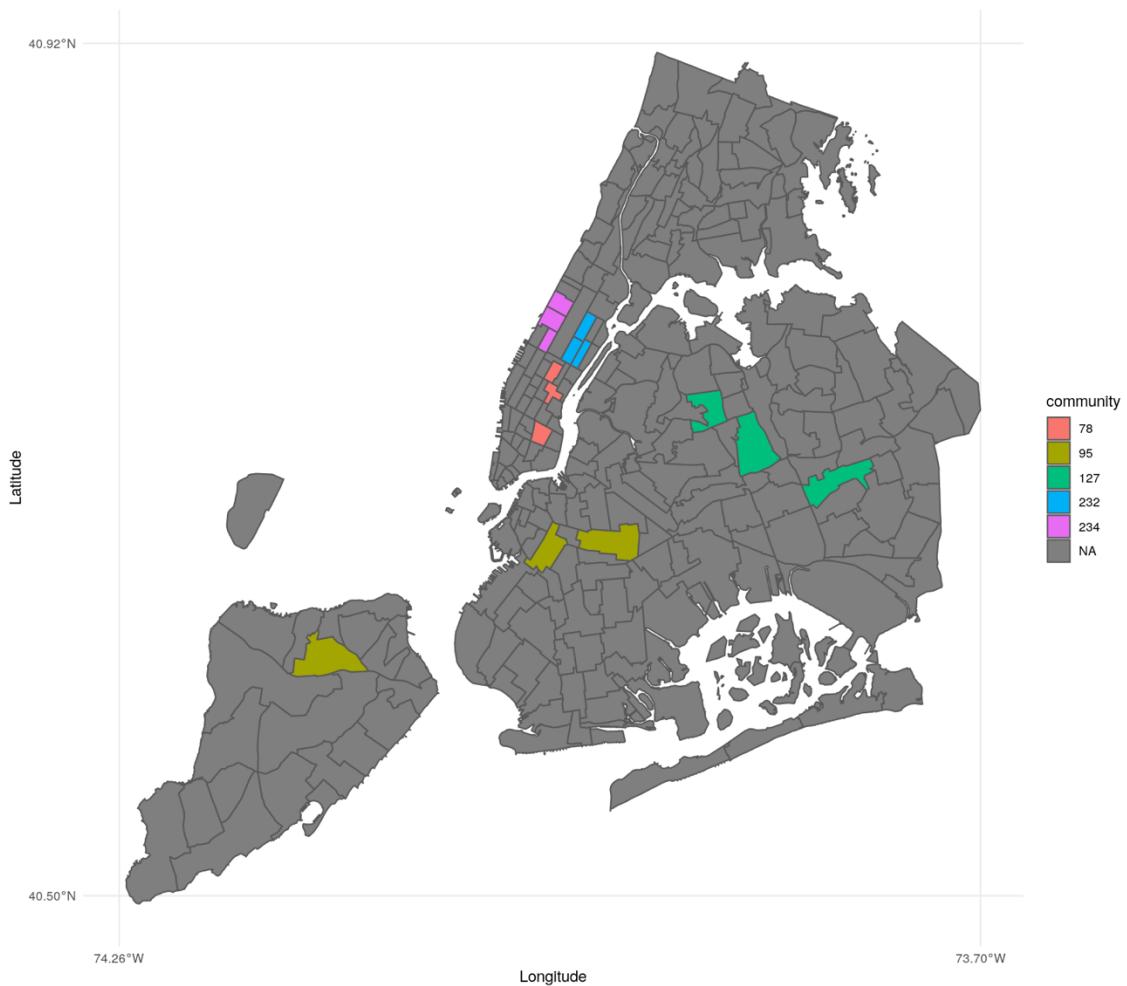


Figure 3.5

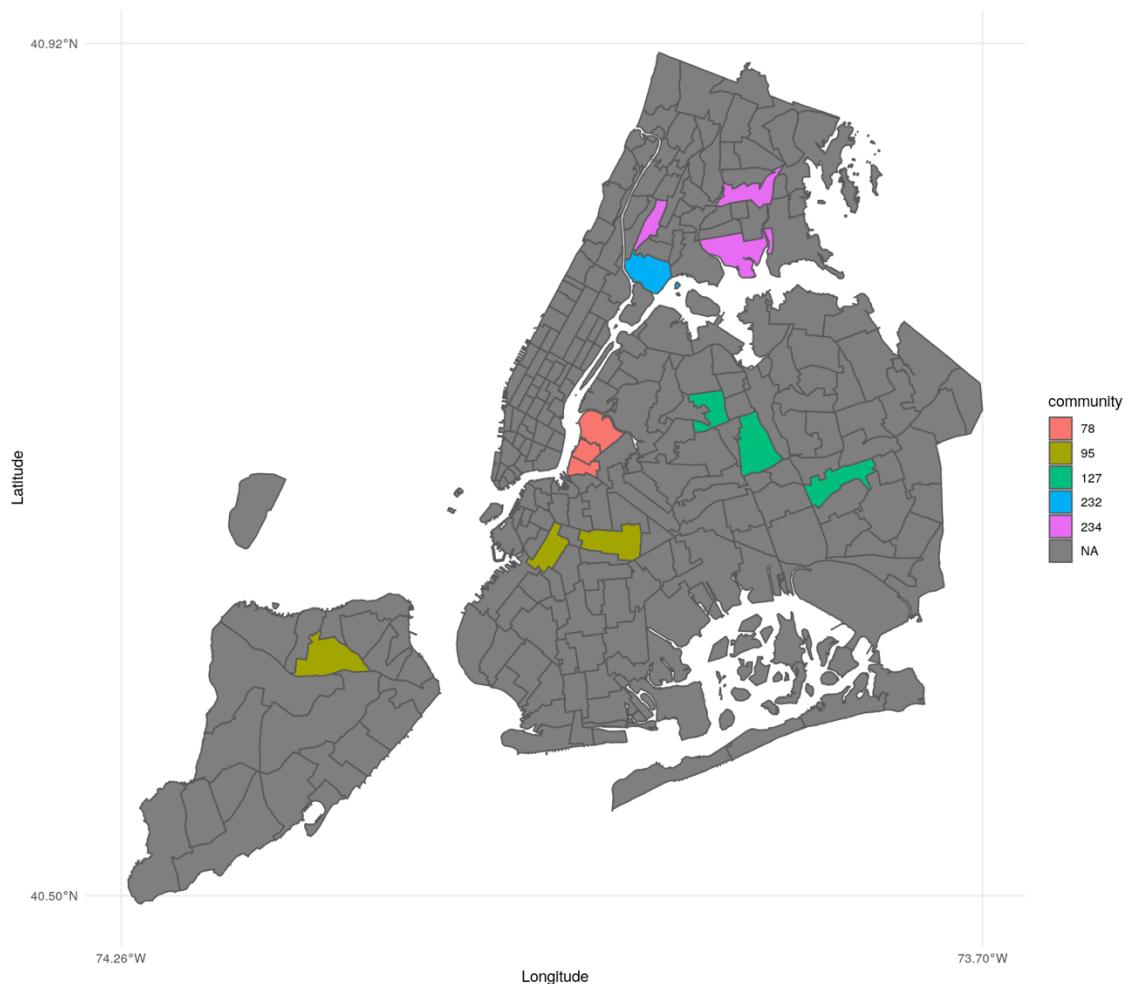


Figure 3.6