DSBA-6520 Final Report

Nicholas Occhipinti
GitHub: https://github.com/nick-occ/dsba-6520-project
Presentation: Finding Clusters of Complaints Presentation

Table of Contents

Problem & Use Case

A project is being funded by the NYPD to analyze data published by NYC Open Data, which contains both NYPD historic complaint data and complaint data for 2021. The data includes felony, misdemeanor and violation crimes that have been reported by the NYPD. Each record in this dataset is a complaint that has occurred between 2006 to 2021. The dataset is large, containing over 7 million records. There are 35 features in the dataset containing information about the type of complaint/offense, the level of offense (felony, misdemeanor, etc), the police jurisdiction and the location (latitude/longitude) of the complaint/offense.

The NYPD needs a way to be able to use this data to develop a tool to identify patterns of complaints happening in different areas of NYC and show the patterns geographically on a map. They also need the tool to be flexible enough to analyze different offenses over the course of different periods of time. Since New York City has such a large population of people and a large area for police to patrol, it is important to spot trends of offenses to better allocate police resources. Time, location and the type of offense are critical variables as they answer the questions, when, where and what is happening in NYC. Having these tools will give them an advantage in knowing where new patterns of offenses are emerging and come up with a plan of action to reduce the numbers.

A Crime Analyst working for New York City has been tasked to work on this project. They determine that for the purposes of the analysis the data will be filtered by more recent dates, ranging between 2019 and 2021. They will also include only the more serious types of offenses, that being felonies. The project scope is only concerned with the NYPD, so other outside entities such as Transit Police will also be excluded. This reduced the dataset from 7.4 million complaints to just under 270,000 records. The complaints occurred in 77 precincts within the 5 boroughs of New York City. The complaints in the dataset represent 65 categories that are more general classifications of offenses that can further be broken down into 369 more specific classifications.

The main goal of the NYPD is to use the tools and analysis results to make better decisions and develop actionable items to help reduce crime in a timely manner. Some examples of improvements that this project could lead to is knowing where to increase the police presence in different areas, better communication between NYPD and communities about emerging crime patterns and identifying where to allocate funds towards technology such as adding more security cameras.

An additional use case the NYPD is interested in studying is how crime has changed during the COVID-19 pandemic. This was also one of the factors for the chosen date range of the data being from 2019 to 2021. This represents three different periods: pre-pandemic (2019), pandemic (2020) and vaccine/return to normal period (2021). The objective is that by analyzing these time periods, conclusions can be derived about if certain types of offenses are higher or lower than normal which could be a result of different factors during the pandemic such as

restaurants\public places being closed, reduction of tourists and people being quarantined in their homes.

Dataset and Data Model

The data model was broken into 5 node types, Complaint, PrecinctName, Location, GeneralOffense and SpecificOffense. The Complaint node uses the “complaint_date” property which is when the complaint was first reported and uses the "complaint_id" property for the constraint to define uniqueness. The date is key to determining how complaints have changed over time.

The PrecinctName node is the precinct where the complaint was reported and is represented by the “precinct” property. The relationship between Complaint and PrecinctName is a directional relationship where the precinct reports a complaint. There are 77 precincts, grouped into one of the following patrol boroughs: Brooklyn North or South, Queens North or South, Bronx, Manhattan North or South and Staten Island. For some of the analysis it is important to know the amount of area each precinct patrols in order to normalize the data. Data from NYC Open Data was used to capture the area and add it to the data model.

The Location node represents where the complaints occur and are spatially created as points through the latitude and longitude properties, this is the smallest level of granularity in this dataset. There is a directional relationship, where complaints are located at locations. Using Geographic Information Systems (GIS) software different analyses were performed to get information assigned to the locations dataset before it was loaded into Neo4J. A "boro_name" property was created by taking a polygon spatial file of NYC Boroughs from NYC Open Data and calculating the borough for all the location points within the 5 polygon features.

The second GIS related analysis performed was getting the nearby distances of location nodes. A unique id was created to make it easier to link the data between the Jupyter notebook and the GIS software, so as a result a "location_id" property was added to the **Location** node. The location data in the Jupyter notebook was imported into the GIS and XY points were generated to display on the map. The Generate Near Table tool calculated the distance for each location node to every other location node within 1000 feet. The output generated “to node” and “from node” fields of location ids and the distance and ranking between the nodes. The output was exported to a CSV and loaded into Neo4J as seen in the LOCATED_NEARBY relationship of the updated data model below along with the "near_distance" and "near_rank" properties of the relationship. More details about the GIS analysis that was performed can be found in Appendix B - GIS Analysis.

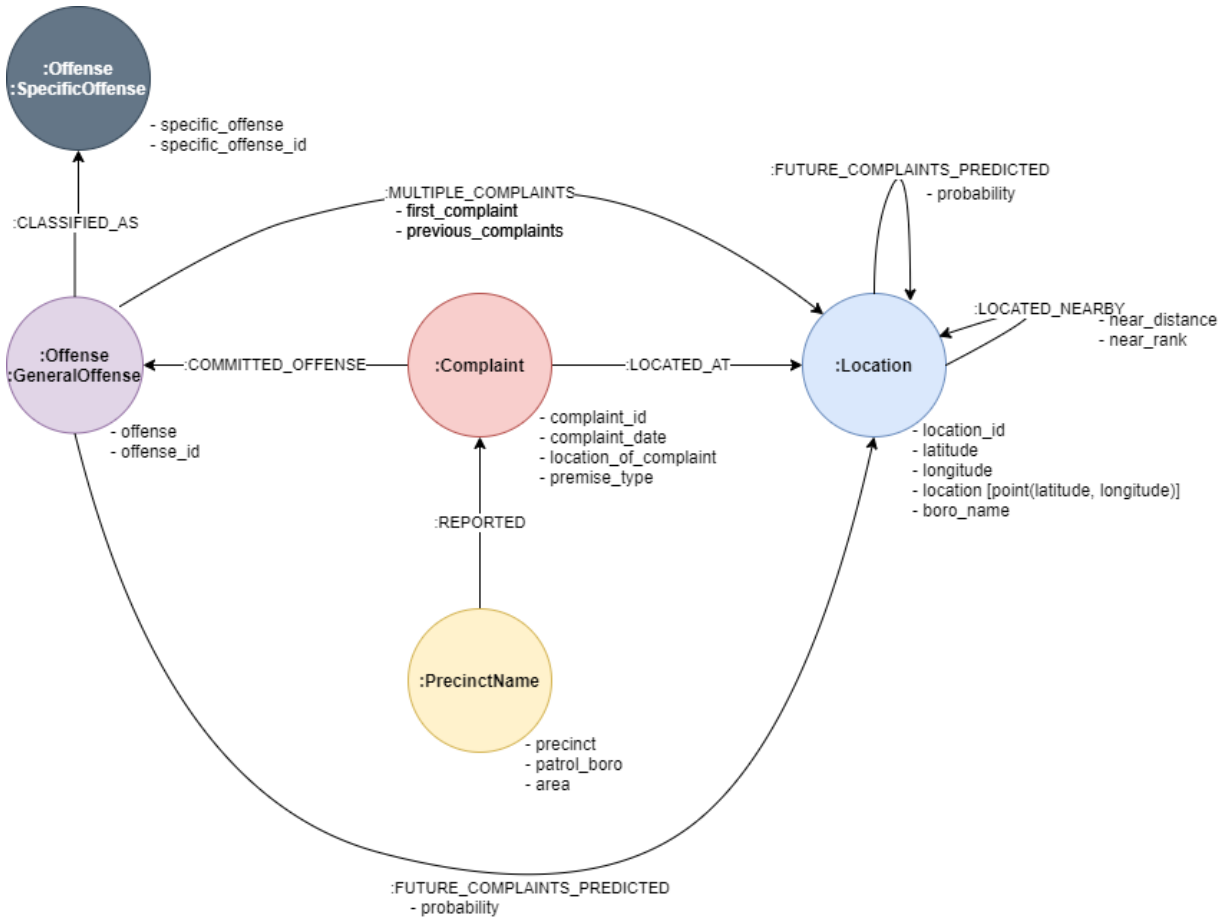
There are two types of offenses, those that are more general and another category of more specific offenses which are broken up into two nodes, GeneralOffense and SpecificOffense respectively. The two offenses are grouped together with an Offense label. Based on the depth

of the analysis the user could just work with the GeneralOffense node, but if more information is needed, the SpecificOffense node offers that extra level of detail. Each offense node has an id number which is the offense code, and a description of what that offense is. There is a directional relationship between complaints where the related complaints contain details of an offense that was committed, and the general offense can get the classification of the specific offense category.

For the purposes of link prediction two relationships were added to the data model from the Offense node label to the Location node label. The MULTIPLE_COMPLAINTS relationship is created when there is more than one complaint at the location. The properties “first_complaint” stores the date of the first complaint and “previous_complaints” as the name implies store a count of previous complaints. The FUTURE_COMPLAINTS_PREDICTED is the link prediction relationship and has the probability as a property. The results are both a bipartite projection between Offense and Location and a monopartite projection between the Location nodes.

Below is the final data model, previous data models can be found in Appendix A - Graph Data Model Revisions.

Final Data Model



Preprocessing

The exploratory data analysis and preprocessing code to clean the data can be found on the eda.ipynb Jupyter notebook on GitHub. Both the year to date (2021) data and the historical data were loaded and then concatenated together into one data frame. The data was filtered to only include data for 2019 to 2021, felony offenses and complaints related to the NYPD.

```
clean_df = complaints.loc[(complaints['LAW_CAT_CD'] == 'FELONY') &
            (complaints['JURIS_DESC'] == 'N.Y. POLICE DEPT') &
            ((complaints['complaint_date'] >= '2019-01-01') & (complaints['complaint_date'] <=
'2021-07-01'))
           ]
```

After examining the data it was determined not all fields were needed so the fields not needed were dropped from the data frame. The fields that were going to be brought into Neo4J were renamed to make the names more meaningful as to what they represent and also to follow the Neo4J naming standards.

```
# remove columns that aren't needed
try:
    clean_df = clean_df.drop(columns=[
        'CRM_ATPT_CPTD_CD',
        'HADEVELOPT',
        'HOUSING_PSA',
        'JURISDICTION_CODE',
        'JURIS_DESC',
        'PARKS_NM',
        'STATION_NAME',
        'SUSP_AGE_GROUP',
        'SUSP_RACE',
        'SUSP_SEX',
        'TRANSIT_DISTRICT',
        'VIC_AGE_GROUP',
        'VIC_RACE',
        'VIC_SEX',
        'LAW_CAT_CD',
        'RPT_DT',
        'X_COORD_CD',
        'Y_COORD_CD',
        'CMPLNT_FR_DT',
        'CMPLNT_FR_TM',
        'CMPLNT_TO_DT',
        'CMPLNT_TO_TM',
        'Lat_Lon',
        'BORO_NM'
    ], axis='columns')
except:
    pass
```

```
# rename columns
rename_dict = {
    "CMPLNT_NUM": "complaint_id",
    "ADDR_PCT_CD": "precinct",
    "CMPLNT_FR_DT": "complaint_from_date",
    "CMPLNT_FR_TM": "complaint_from_time",
    "CMPLNT_TO_DT": "complaint_to_date",
    "CMPLNT_TO_TM": "complaint_to_time",
    "KY_CD": "offense_id",
    "LOC_OF_OCCUR_DESC": "location_of_complaint",
    "OFNS_DESC": "offense",
    "PATROL_BORO": "patrol_boro",
    "PD_CD": "specific_offense_id",
    "PD_DESC": "specific_offense",
    "PREM_TYP_DESC": "premises_type",
    "Latitude": "latitude",
    "Longitude": "longitude"
}

clean_df = clean_df.rename(rename_dict, axis="columns")
```

A function was created to fill in the missing data for fields by searching other records that are populated with the field and if there is only 1 value then it will replace the missing value, otherwise it

is marked as UNKNOWN. An example is a record with a missing patrol boro, the function will check the precinct associated with the missing patrol boro and check if other records for that precinct have a patrol boro filled in, and if there is only one other patrol boro value for that precinct that will replace the missing value.

```
# function to take missing data and lookup other the missing values from a common search field
def fill_missing_data(search_fld_vals, search_fld, missing_fld):
    l = list(clean_df.loc[(clean_df[search_fld] == search_fld_vals) &
                          ~(clean_df[missing_fld].isna())][missing_fld].unique())
    if len(l) == 1:
        clean_df.loc[(clean_df[search_fld] == search_fld_vals) &
                     (clean_df[missing_fld].isna()), missing_fld] = l[0]
        return l[0]
    clean_df.loc[(clean_df[search_fld] == search_fld_vals) &
                 (clean_df[missing_fld].isna()), missing_fld] = 'UNKNOWN'
    return 'UNKNOWN'
```

For complaints, the dates were checked that they were from 2019 - 2021 and for the premises_type and location_of_complaint fields, missing values were replaced with UNKNOWN. For the precinct related data, the word 'Precinct ' was prepending the precinct field primarily for visualizations so it would be clear that the number represents a precinct. The patrol borough called the "fill_missing_data" function and used the precinct to determine the valid value. The area was also taken from an external precincts file and merged into the dataset.

```
# merged precincts and precincts area data frames
precincts_merged = precincts.merge(precinct_area, how='outer', on='precinct')
precincts_merged
```

	complaint_id	precinct	patrol_boro	area
0	294801329	Precinct 75	BKLYN NORTH	1.803843e+08
1	444277080	Precinct 75	BKLYN NORTH	1.803843e+08
2	415582914	Precinct 75	BKLYN NORTH	1.803843e+08
3	896161140	Precinct 75	BKLYN NORTH	1.803843e+08
4	413986979	Precinct 75	BKLYN NORTH	1.803843e+08
...	...	...	...	...
270216	215729359	Precinct 78	BKLYN SOUTH	6.767711e+07
270217	731445070	Precinct 78	BKLYN SOUTH	6.767711e+07
270218	959502021	Precinct 78	BKLYN SOUTH	6.767711e+07
270219	520929376	Precinct 78	BKLYN SOUTH	6.767711e+07
270220	227897511	Precinct 78	BKLYN SOUTH	6.767711e+07

270221 rows × 4 columns

For the offenses, similar offenses were cleaned up to only use one value.

```
# replace duplicate category
clean_df.loc[clean_df.offense == "KIDNAPPING AND RELATED OFFENSES", 'offense'] =
'KIDNAPPING & RELATED OFFENSES'
```

Missing offenses were run through the “fill_missing_data function” to either assign it an offense or a default value of UNKNOWN or if the id was missing a default id of 0.

```
[fill_missing_data(o, 'offense_id', 'offense') for o in missing_offense]
```

```
['MISCELLANEOUS PENAL LAW', 'UNKNOWN']
```

For the locations, a location_id field was created to have a unique id for every unique location. This was needed for the GIS analysis that was performed and detailed in Appendix B - GIS Analysis.

```
unique_locs.rename({'index':'location_id'}, axis='columns', inplace=True)
```

```
unique_locs
```

	location_id	latitude	longitude
0	0	40.499948	-74.238006
1	1	40.500216	-74.244795
2	2	40.500563	-74.243049
3	3	40.500629	-74.236556
4	4	40.500918	-74.241253

The data was then exported to the GIS so that it can be loaded as points using the latitude and longitude, the first analysis was capturing the boro using a polygon spatial file based on where the points fell inside the polygon. The data was exported and merged into the location dataset.

```
locations_merged = locations.merge(unique_locs_with_boro, how='outer', on=['latitude','longitude'])
```

```
locations_merged
```

	complaint_id	latitude	longitude	location_id	boro_name
0	294801329	40.668188	-73.875137	17588	Brooklyn
1	984985109	40.668188	-73.875137	17588	Brooklyn
2	236124378	40.668188	-73.875137	17588	Brooklyn
3	403601885	40.668188	-73.875137	17588	Brooklyn
4	210506500	40.668188	-73.875137	17588	Brooklyn
...	...	...	...	...	...
270216	618850936	40.787229	-73.811050	48269	Queens
270217	461353346	40.769837	-73.769712	46159	Queens
270218	177906577	40.692858	-73.903543	26082	Brooklyn
270219	877569272	40.662082	-73.861913	15968	Brooklyn
270220	463193037	40.692865	-73.796835	26084	Queens

270221 rows × 5 columns

The second analysis involved generating a near table for any locations within 1000ft from one another. The output contained an in (from) and near (to) location id along with the distance and a ranking for the top near locations for the given in location. The results were exported and used to create the “LOCATED_NEARBY” relationship between locations.

```
locations_near = pd.read_csv('./data/gis/output/locations_near.csv', usecols=['in_location_id','near_location_id','NEAR_DIST','NEAR_RANK'])
```

```
locations_near.rename({
    "NEAR_DIST": "near_dist",
    "NEAR_RANK": "near_rank"
}, axis='columns', inplace=True)
```

```
locations_near
```

	near_dist	near_rank	in_location_id	near_location_id
0	473.523635	1	0	3
1	571.947999	2	0	7
2	629.578843	3	0	6
3	774.973099	4	0	5
4	853.867313	5	0	10
...	...	...	...	...
2339983	523.699329	1	61513	61512
2339984	526.376203	2	61513	61511
2339985	611.923327	3	61513	61510
2339986	756.534984	4	61513	61509
2339987	976.969304	5	61513	61502

2339988 rows × 4 columns

```
# export to csv
locations_near.to_csv('./data/imports_neo4j/locations_near.csv', index=None)
```

Database Setup

After the preprocessing was complete and the dataset was cleaned the data for each node type was exported to a separate CSV file to load into Neo4J. The details of the database setup is in the neo4j_loading.ipynb Jupyter notebook. There were 6 CSV files in total, 5 for the node types and 1 for the LOCATED_NEARBY relationship between the Location nodes.

A Neo4J project called" NYPD Complaints REV1" was created and a new database was set up under the neo4j username. The following constraints were run to preserve uniqueness of each node before the data was loaded:

CREATE CONSTRAINT unique_complaint ON (n:Complaint) ASSERT n.complaint_id IS UNIQUE

CREATE CONSTRAINT unique_precinct ON (n:PrecinctName) ASSERT n.precinct IS UNIQUE

CREATE CONSTRAINT unique_location ON (n:Location) ASSERT n.location_id IS UNIQUE

CREATE CONSTRAINT unique_general_offense ON (n:GeneralOffense) ASSERT n.offense IS UNIQUE

CREATE CONSTRAINT unique_specific_offense ON (n:SpecificOffense) ASSERT n.specific_offense IS UNIQUE

Complaint Node

```
// Load Complaint nodes
LOAD CSV WITH HEADERS FROM "file:///complaints.csv" as row
CREATE (c:Complaint {
    complaint_id: row.complaint_id,
    location_of_complaint: row.location_of_complaint,
    premises_type: row.premises_type,
    complaint_date: row.complaint_date
})
```

PrecinctName Node

```
// Load PrecinctName nodes and create REPORTED relationship with Complaint
LOAD CSV WITH HEADERS FROM "file:///precincts.csv" as row
MERGE (p:PrecinctName {
    precinct: row.precinct,
    patrol_boro: row.patrol_boro,
    area: row.area})
WITH row, p
MATCH(c:Complaint {complaint_id: row.complaint_id})
MERGE (p)-[:REPORTED]->(c)
```

Location Node

```
// Load Location node and create LOCATED_AT relationship with Complaint
LOAD CSV WITH HEADERS FROM "file:///locations.csv" as row
MERGE (l:Location {
location: point({
latitude:toFloat(row.latitude),
longitude:toFloat(row.longitude)}),
latitude: row.latitude,
longitude: row.longitude,
location_id: row.location_id,
boro_name: row.boro_name
})
WITH row, l
MATCH(c:Complaint {complaint_id: row.complaint_id})
MERGE (c)-[:LOCATED_AT]->(l)
```

```
LOAD CSV WITH HEADERS FROM "file:///locations_near.csv" AS row
//look up the two nodes we want to connect up
MATCH (l1:Location {location_id:row.in_location_id}),(l2:Location {location_id:row.near_location_id})
//now create a relationship between them
CREATE (l1)-[:LOCATED_NEARBY {
near_distance: row.near_dist,
near_rank: row.near_rank
}]->(l2);
```

Offenses Node

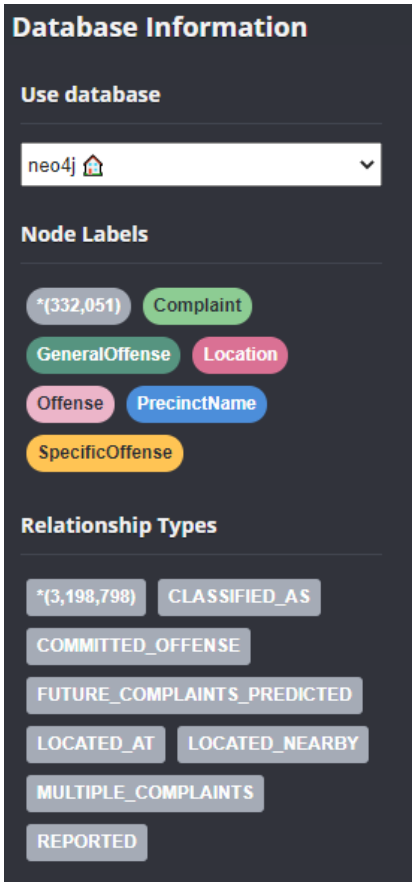
```
// Load Offense:GeneralOffense node and create COMMITTED_OFFENSE relationship with Complaint
LOAD CSV WITH HEADERS FROM "file:///general_offenses.csv" as row
MERGE (o:Offense:GeneralOffense {
    offense: row.offense,
    offense_id: row.offense_id

})
WITH row, o
MATCH(c:Complaint {complaint_id: row.complaint_id})
MERGE (c)-[:COMMITTED_OFFENSE]->(o)
```

```
LOAD CSV WITH HEADERS FROM "file:///specific_offenses.csv" as row
MERGE (s:Offense:SpecificOffense {
    offense: row.specific_offense,
    offense_id: row.specific_offense_id

})
WITH row, s
MATCH(o:Offense:GeneralOffense {offense_id: row.offense_id})
MERGE (o)-[:CLASSIFIED_AS]->(s)
```

This produced a Neo4J database with the following Node Labels and Relationship Types:



Database Queries

Precinct Complaints in 2021

This query captures a high level snapshot of the year to date complaints to see which precincts are dealing with the most complaints. This information will help when comparing if the hotspots of rising complaints are occurring in the precincts with the most complaints. The data is also normalized to account for the area of each precinct and the number is sorted by the “complaints_per_100k_sqft” field. One initial finding in the data below shows that 9 out of the top 10 precincts are in Manhattan.

```
MATCH(p:PrecinctName)-[:REPORTED]->(c:Complaint)
where c.complaint_date >= "2021-01-01"
return p.precinct as precinct, p.patrol_boro as patrol_boro, count(c) as number_of_complaints, p.area as area,
count(c)/p.area * 100000 as
complaints_per_100k_sqft
order by complaints_per_100k_sqft desc
```

precinct	patrol_boro	number_of_complaints	area	complaints_per_100k_sqft
Precinct 14	MAN SOUTH	599	20510163.83	2.920503
Precinct 28	MAN NORTH	251	15289544.60	1.641645
Precinct 46	BRONX	531	38323373.38	1.385577
Precinct 33	MAN NORTH	343	25865037.87	1.326114
Precinct 32	MAN NORTH	282	23009990.36	1.225555
Precinct 6	MAN SOUTH	264	22098189.76	1.194668
Precinct 9	MAN SOUTH	255	21394233.59	1.191910
Precinct 18	MAN SOUTH	374	32261097.60	1.159291
Precinct 30	MAN NORTH	217	18845037.81	1.151497

Number of Complaints Grouped by Precinct, Offense and Date

This query is a revision of the “Precinct Complaints in 2021” query, it captures all the complaints by precinct and groups them by the complaint date in “yyyy-MM” format. It also adds the offense in the query and displays the area and calculates the number of complaints per 100K square feet. This query is used to fetch information in the choropleth map described in the Technical Solution of Map Visualizations section.

```
MATCH (o:GeneralOffense)<-[:COMMITTED_OFFENSE]-(c:Complaint)<-[:REPORTED]-(p:PrecinctName)
with p.precinct as precinct, apoc.temporal.format(date(c.complaint_date),"yyyy-MM") as complaint_date, o.offense as offense, apoc.convert.toFloat(p.area) as area, count(c) as complaints
return precinct, complaint_date, offense, complaints, complaints/area  * 100000 as complaints_per_100k_sqft order by precinct
```

precinct	complaint_date	offense	complaints	complaints_per_100k_sqft
Precinct 1	2019-04	GRAND LARCENY	66	0.139488
Precinct 1	2020-08	GRAND LARCENY	61	0.128921
Precinct 1	2020-05	CRIMINAL MISCHIEF & RELATED OF	21	0.044383
Precinct 1	2020-01	GRAND LARCENY	94	0.198665
Precinct 1	2020-08	BURGLARY	14	0.029588

Top 5 Offenses per Precinct in 2021

This query gets the top 5 offenses reported by the precincts in 2021. It is not only important to understand the quantity of offenses but also what those offenses are. Part of this project is

being able to give the NYPD feedback that they can use to make better decisions, and different offenses require different plans of actions. One pattern found in the example below is that Grand Larceny appears to be a common offense in most precincts. This query provided valuable information about offenses to target as part of the analysis for the graph data science algorithms and visualizations.

```
MATCH(p:PrecinctName)-[:REPORTED]->(c:Complaint)-[:COMMITTED_OFFENSE]-(o:Offense)
where c.complaint_date >= "2021-01-01"
with p.precinct as precinct, p.area as area, o.offense as offense, count(c) as complaints
order by precinct, complaints desc
with precinct, area, offense, complaints, complaints/area * 100000 as complaints_per_100k_sqft
order by complaints desc
with precinct, collect({offense:offense, complaint:complaints,complaints_per_100k_sqft:complaints_per_100k_sqft,
area: area})[..5] as top_5_offenses unwind top_5_offenses as o
return precinct, o.offense as offense, o.complaint as number_of_complaints, o.area as area,
o.complaints_per_100k_sqft as complaints_per_100k_sqft
order by precinct, complaints_per_100k_sqft desc
```

precinct	offense	number_of_complaints	area	complaints_per_100k_sqft
Precinct 1	GRAND LARCENY	143	4.731589e+07	0.302224
Precinct 1	BURGLARY	27	4.731589e+07	0.057063
Precinct 1	FELONY ASSAULT	25	4.731589e+07	0.052836
Precinct 1	ROBBERY	21	4.731589e+07	0.044383
Precinct 1	CRIMINAL MISCHIEF & RELATED OF	20	4.731589e+07	0.042269
Precinct 10	GRAND LARCENY	75	2.726732e+07	0.275055
Precinct 10	BURGLARY	35	2.726732e+07	0.128359
Precinct 10	CRIMINAL MISCHIEF & RELATED OF	21	2.726732e+07	0.077015
Precinct 10	ROBBERY	20	2.726732e+07	0.073348
Precinct 10	DANGEROUS WEAPONS	11	2.726732e+07	0.040341

Difference in complaints between 2019 and 2020

It is important to understand how complaints have changed over time to determine emerging patterns of offenses. The dates for this project cover 2019, 2020, and 2021. Since 2019 and 2020 represent a full years' worth of data, this query compares the two years number of complaints and calculates the difference to see if complaints have risen, declined or stayed the same for each precinct. The other part of the use case is understanding how complaints have changed because of the pandemic, years 2019 and 2020 also represent a pre-pandemic and pandemic comparison.

```
MATCH (p:PrecinctName)-[:REPORTED]->(c:Complaint)
with p.precinct as precinct, p.patrol_boro as patrol_boro, date(c.complaint_date).year as year, count(c) as complaints
order by year
```

```
where year in [2019,2020]
with precinct, patrol_boro, collect(year) as years, collect(complaints) as complaints_per_year
return  precinct, patrol_boro, years, complaints_per_year, complaints_per_year[1] - complaints_per_year[0] as difference_19_20
order by difference_19_20 desc
```

precinct	patrol_boro	years	complaints_per_year	difference_19_20
Precinct 43	BRONX	[2019,2020]	[2286,2654]	368
Precinct 104	QUEENS NORTH	[2019,2020]	[1727,1969]	242
Precinct 102	QUEENS SOUTH	[2019,2020]	[1229,1469]	240
Precinct 48	BRONX	[2019,2020]	[1970,2145]	175
Precinct 94	BKLYN NORTH	[2019,2020]	[1030,1203]	173
Precinct 33	MAN NORTH	[2019,2020]	[1042,1209]	167
Precinct 46	BRONX	[2019,2020]	[2477,2635]	158
Precinct 42	BRONX	[2019,2020]	[1768,1916]	148
Precinct 103	QUEENS SOUTH	[2019,2020]	[2098,2242]	144
Precinct 83	BKLYN NORTH	[2019,2020]	[2018,2141]	123
Precinct 40	BRONX	[2019,2020]	[2688,2789]	101
Precinct 23	MAN NORTH	[2019,2020]	[825,917]	92

Top 5 Locations per Precinct

This query aims to examine the data at a more granular level; the location of where the complaints are occurring. It provides some initial insight into potential hotspots of where complaints could be formed and can be used to verify the results of more advanced graph data science algorithms that attempt to group similar nodes into communities. This query is relevant to the project because a key part is being able to identify areas that will help show precincts where there are a high number of offenses reported.

```
match(p:PrecinctName)-[:REPORTED]->(c:Complaint)-[:LOCATED_AT]->(l:Location)
with p.precinct as precinct, count(c) as complaints, l.location as location
order by precinct, complaints desc
with precinct, collect({location:location,complaint:complaints})[..5] as top_5_locations unwind top_5_locations as l
return precinct,  l.location as location, l.complaint as number_of_complaints
order by precinct, number_of_complaints desc
```

precinct	location	number_of_complaints
Precinct 1	point({srid:4326, x:-74.00709028, y:40.72025522})	64
Precinct 1	point({srid:4326, x:-74.01060963, y:40.71009385})	56
Precinct 1	point({srid:4326, x:-73.99985714, y:40.72441101})	48
Precinct 1	point({srid:4326, x:-74.01144362, y:40.71461714})	47
Precinct 1	point({srid:4326, x:-74.00165018, y:40.72355738})	41
...	...	...
Precinct 94	point({srid:4326, x:-73.95982681, y:40.71588701})	47
Precinct 94	point({srid:4326, x:-73.95311663, y:40.72696507})	44
Precinct 94	point({srid:4326, x:-73.96278968, y:40.720082})	40
Precinct 94	point({srid:4326, x:-73.95380782, y:40.72917213})	29
Precinct 94	point({srid:4326, x:-73.95287135, y:40.72688262})	25

Graph Data Science Algorithms

The selected algorithms focus on the key questions, what is happening (Offense), when is it happening (Complaint) and and where is it happening (Location). There are a variety of dates and offenses one could look for in the database, which is why a user interface was built into the mapping visualization to give the user the flexibility to choose. As a result the algorithms were parameterized to pass in the selected offenses and date range and generate results when the user clicks a button on the map. The algorithms used can be found in the neo4j_integration.ipynb Jupyter notebook.

Louvain Modularity Algorithm

The Louvain Modularity Algorithm was used for community detection and helps identify clusters of common nodes by iterating over the graph and developing intermediate communities before forming a final community. For the purposes of the analysis common Location nodes were identified. The relationship projection query takes into account the relationship between other nearby locations where complaints occurred and the node query filters locations where common offenses occurred within the user specified date range.

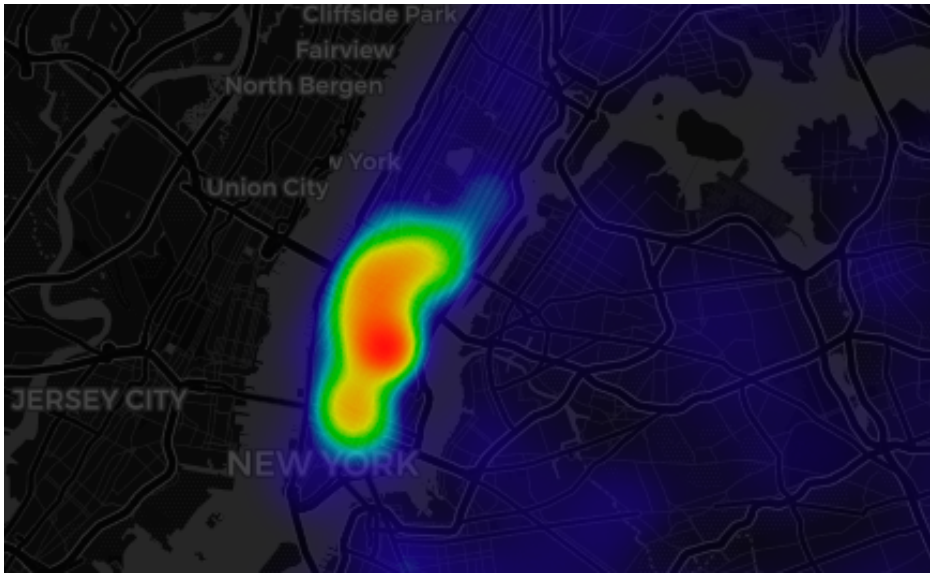
```
louvain_query = 'CALL gds.louvain.stream({  \n\
    nodeQuery: "MATCH (l:Location)<-[:LOCATED_AT]-(c:Complaint)-[:COMMITTED_OFFENSE]->(o:Offense) ' \
    'where o.offense in '+ str(offenses) + ' and c.complaint_date >= ' + from_date + ' and c.complaint_date <= ' +
to_date + '  return distinct id(l) as id",\n\
```

```
    relationshipQuery: "MATCH (c:Complaint)-[:LOCATED_AT]->(l1:Location)-[:LOCATED_NEARBY]->(l2:Location)
return id(l1) as source, id(l2) as target",\n\
    validateRelationships: false,\n\
    maxIterations: 50})\n\
    yield nodeId, communityId\n\
    RETURN gds.util.asNode(nodeId).location_id AS location_id, gds.util.asNode(nodeId).latitude AS latitude,
gds.util.asNode(nodeId).longitude AS longitude, communityId as community'
```

This algorithm is being called inside a function that is called when the user clicks a button on the map. There are three parameters being passed, the list of offenses and the from and to date. Once community ids are assigned to the locations a query is being run to get the number of complaints for these locations based on the selected offenses and date range.

```
number_of_complaints_query = 'MATCH
(l:Location)<-[:LOCATED_AT]-(c:Complaint)-[:COMMITTED_OFFENSE]->(o:Offense) where o.offense in '+
str(offenses) + ' and c.complaint_date >= ' + from_date + ' and c.complaint_date <= ' + to_date + ' return distinct
l.location_id as location_id, count(c) as number_of_complaints'
```

These results are merged with the Louvain community results and clusters are generated by taking the average coordinates in each community and summing the total number of complaints per community. This data can then be generated into a heatmap similar to the one below to show the locations of these offenses and the intensity based on the number of complaints.



PageRank Centrality Algorithm

PageRank was used in order to find the most influential locations that are central to other locations nearby where similar offenses occurred. This can help the NYPD learn where the heart of the offenses are occurring and it could lead to them realigning patrol routes in these areas in an effort to reduce the number of complaints. The algorithm's relationship projection takes into account the offenses and the date range as well as similar offenses located nearby to determine what node to rank highest.

```
page_rank_query = 'CALL gds.pageRank.stream({ \n\
   nodeQuery: "match (l:Location) return distinct id(l) as id",\n \
   relationshipQuery: "match
(o:Offense)<-[:COMMITTED_OFFENSE]-(c:Complaint)-[:LOCATED_AT]->(l:Location)-[:LOCATED_NEARBY]->(l2:Lo
cation) where o.offense in ' + str(offenses) + ' and c.complaint_date >= ' + from_date + ' return id(l) as source, id(l2)
as target",\n\
   validateRelationships: false})\n\
   YIELD nodeId, score\n\
   with gds.util.asNode(nodeId).location_id AS location_id, gds.util.asNode(nodeId).latitude AS latitude,
gds.util.asNode(nodeId).longitude AS longitude, score\n\
   where location_id is not null\n\
   return location_id, latitude, longitude, score\n\
   order by score desc\n\
   limit 10'
```

This algorithm is being called inside a function that is called when the user clicks a button on the map. There are two parameters being passed, the list of offenses and the from date. Only the top 10 rankings are being returned and using the location field they are drawn on a map. The following query is also run after the rankings are returned to get the locations nearby that had similar offenses during the same time frame.

```
nearby_locs_query = 'match
(l1:Location)-[ln:LOCATED_NEARBY]-(l2:Location)<-[:LOCATED_AT]-(c:Complaint)-[:COMMITTED_OFFENSE]->(o:
Offense) \n\
      where l1.location_id in '+ str(list(page_rank_df.location_id.unique())) + 'and c.complaint_date >= ' + from_date + '
and o.offense in '+ str(offenses) + ' with \
      apoc.convert.toFloat(l1.latitude) as l1_lat,apoc.convert.toFloat(l1.longitude) as l1_lon,
apoc.convert.toFloat(l2.latitude) as l2_lat,apoc.convert.toFloat(l2.longitude) as l2_lon \
      return distinct [l1_lat,l1_lon] as l1,[ l2_lat,l2_lon] as l2'
```

Together this allows for both the ranked locations and the nearby locations to be drawn on a map and it will also draw a line connecting the locations together to better visualize why the ranked location was selected. In the example below the red points are the page rank location and the blue are locations nearby with the same offenses.

Label Propagation of Offenses

This algorithm uses Label Propagation to analyze different categories of offenses ranging from general to specific, it is useful to analyze how the offenses are related and how the different node types can be grouped together into communities. Certain groupings can be offenses that are closely related in meaning such as “Forgery,Etc..” and “Forgery”, but the added value come when the algorithm finds common relationships between two different types of offenses such as “Felony Assault” and “Robbery” and could provide valuable information on patterns of crime and how they are correlated. The previous two algorithms address the where, in terms of where patterns of high offenses are occurring while this algorithm aims to answer the what, as in what patterns of offenses are occurring.

```
CALL gds.graph.create('nypd-comp-labelprop', 'Offense', 'CLASSIFIED_AS')
CALL gds.labelPropagation.stream('nypd-comp-labelprop')
YIELD nodeId, communityId AS community_id
RETURN gds.util.asNode(nodeId).offense AS offense, community_id
ORDER BY community_id, offense
```

offense	community_id
FORGERY	269839
FORGERY,ETC.,UNCLASSIFIED-FELO	269839
ASSAULT POLICE/PEACE OFFICER	269840
FELONY ASSAULT	269840
ROBBERY	269840

Technical Solution of Map Visualizations

For the final deliverable to the NYPD, they wanted an interactive map to show the results to analyze patterns geographically. The maps were integrated into the Jupyter notebook through the use of two Python modules, “ipyleaflet” and “ipywidgets”. The “ipyleaflet” module is the mapping component and “ipywidgets” module controls the user interface widgets.

The value of this type of solution is that it goes beyond looking at just nodes and edges, and allows you to bring in different layers such as a street basemap and overlay it with your results. This combination gives you a point of reference to identify where in NYC the results are being shown. Also these visualizations directly call the data science algorithms which are parameterized to accept the offenses and date range selected by the user.

The visualizations focus on some of the offenses with the most complaints shown in the query results below.

match (c:Complaint)-[:COMMITTED_OFFENSE]-(o:Offense) return o.offense as offense, count(c) as number_of_complaints order by number_of_complaints desc

offense	number_of_complaints
"GRAND LARCENY"	78517
"FELONY ASSAULT"	38555
"BURGLARY"	27879
"ROBBERY"	25727

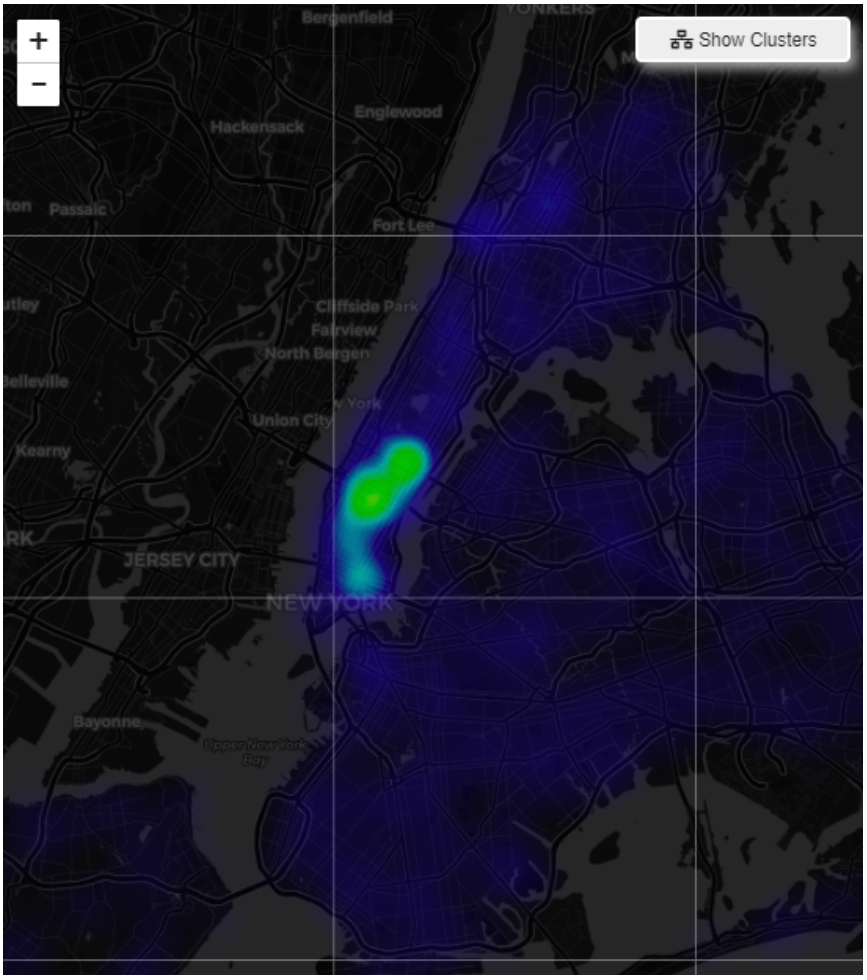
Hotspot Cluster Analysis using Louvain Communities

Implementation

For the hotspot cluster analysis, clusters were generated on a button click event, which takes the selected offenses and the date range set in the user interface widgets and calls the "calculate_clusters" method located in the neo4j_integration.ipynb notebook. The details of the algorithm executed are described in the **Louvain Modularity Algorithm.** The last part of the method is where the layer is created as a Heat Map layer added to the map object "cluster_map".

```
cluster_map.add_layer(Heatmap(locations=lat_lon, radius=20))
```

The layer is displayed on the map, similar to the example below. The higher intensity (green) identifies the larger concentration of complaints and identifies which areas are being affected by the selected offenses during the selected time period.

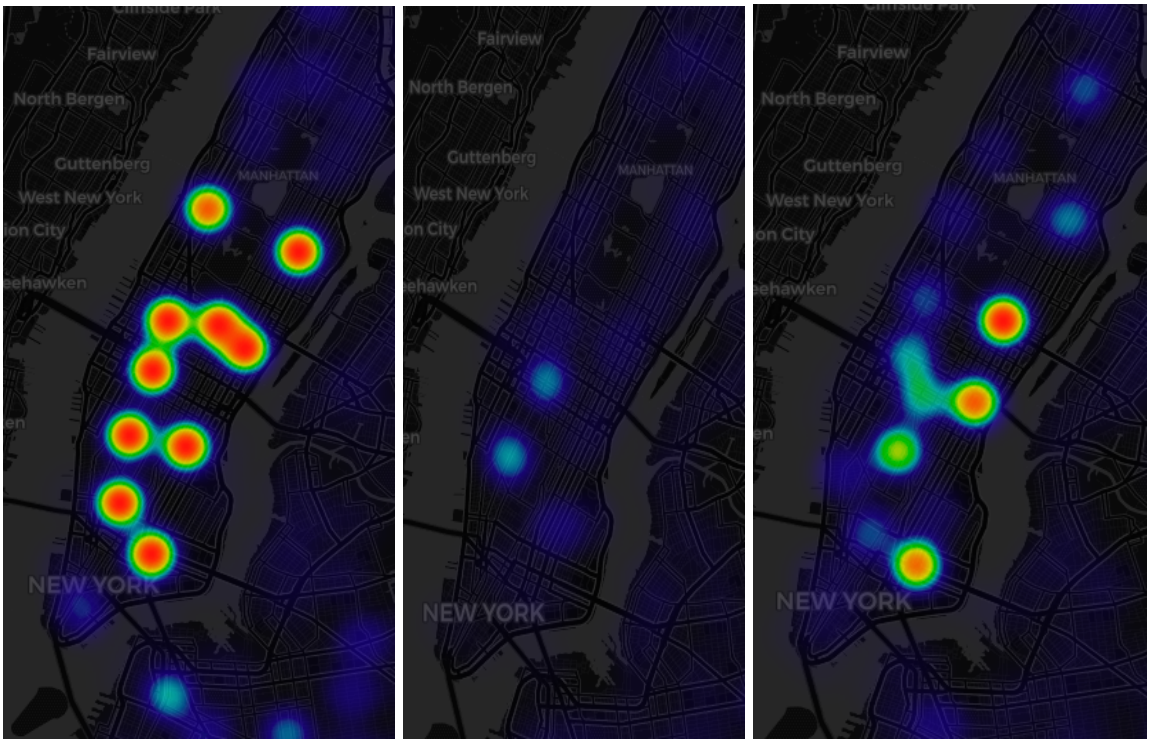


Value Added

To demonstrate the value this type of visualization adds, the number one offense committed (Grand Larceny) was examined to analyze how the offense has changed over time.

Grand Larceny Comparison

4/1/2019 - 4/30/2019	4/1/2020 - 4/30/2020	3/1/2021 - 3/30/2021

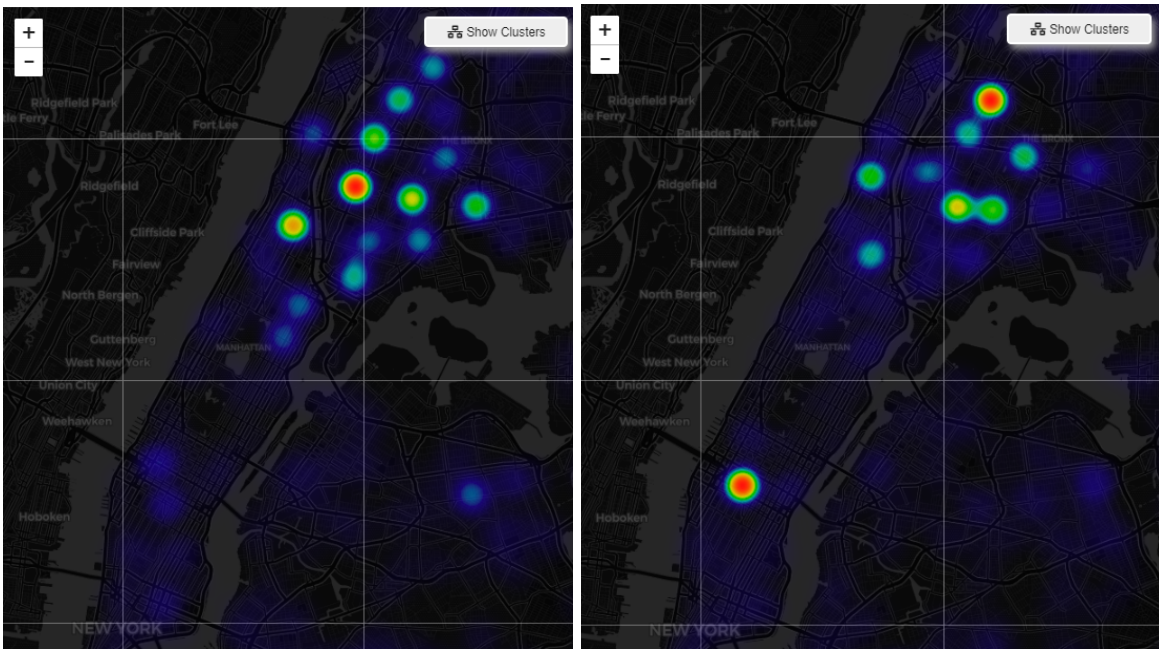


The results show a big drop in grand larceny complaints between 2019 and 2020 and we can see an emergence of hotspots appearing in 2021. The dates also represent different time periods of the pandemic, 2019 being pre-pandemic, 2020 pandemic and 2021 returning to “normal”. These results could also be an indicator to how offenses changed dramatically during the pandemic. An article from June 2021 in the New York Post (https://nypost.com/2021/06/08/grand-larcenies-rebound-as-nyc-opens-back-up/) also mentions the trend of rising grand larceny complaints as the city reopens and note that 2020 had the lowest number of grand larceny complaints in 20 years.

Another example is Felony Assault which is the second most reported offense. Looking into more recent offenses in a two month window from 12/1/20 - 1/31/21 we see that most felony assaults occurred in upper Manhattan and the Bronx. Jump ahead to 2/1/21 - 3/30/21 we see similar activity in that area but also an increase in midtown Manhattan. What was once blue two months ago, now has a higher intensity indicating an increase in complaints.

Felony Assault

12/1/20 - 1/31/21	2/1/21 - 3/30/21



Comparison of Precinct Complaints by Dates

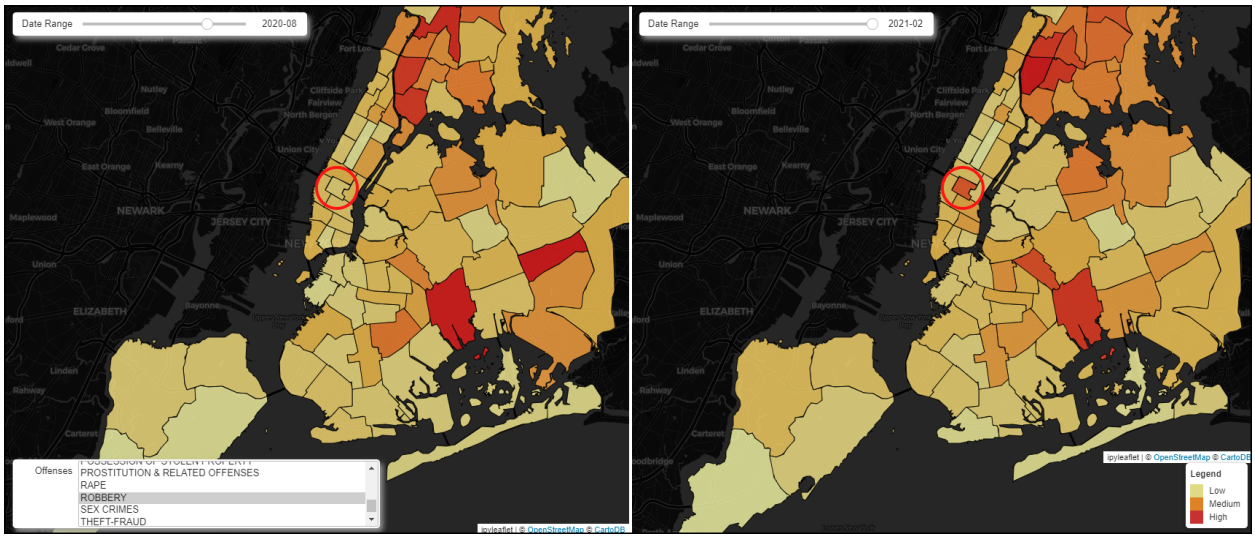
Implementation

For the precinct comparison analysis, the following query was executed to capture the number of complaints by precinct, date (MM-YYYY) and offense.

```
MATCH (o:GeneralOffense)<-[:COMMITTED_OFFENSE]-(c:Complaint)<-[:REPORTED]-(p:PrecinctName)
with p.precinct as precinct, apoc.temporal.format(date(c.complaint_date),"yyyy-MM") as complaint_date, o.offense as offense, apoc.convert.toFloat(p.area) as area, count(c) as complaints
RETURN precinct, complaint_date, offense, complaints, complaints/area  * 100000 as complaints_per_100k_sqft
order by precinct
```

On a button click event, the “compare_complaint_dates” method is called, located in the neo4j_integration.ipynb notebook. The data is filtered by the dates and offenses selected and the layers are created by joining the results with the GeoJson Precinct layer. The layers are added to the map objects, displaying the new choropleth maps.

The layers are displayed on the map, similar to the example below. The red values indicate a high number of complaints, orange represent medium levels and yellow are low levels.

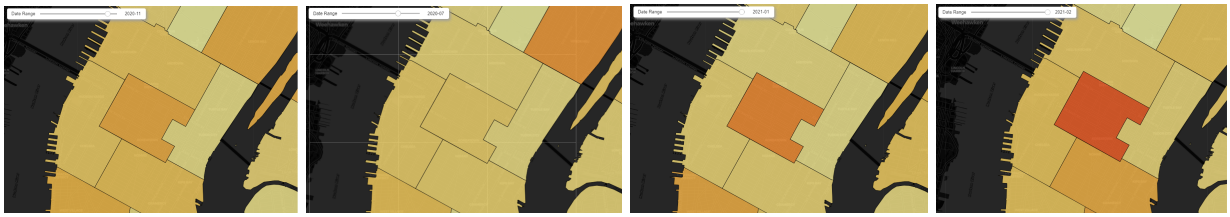


Value Added

In the example above when precincts are compared one precinct stands out (Precinct 14) as having a transition from a low complaint area to a high increase in robberies in a seventh month period of time. To determine if this is a one off spike or an emerging pattern we can compare the months in between.

Robberies - Precinct 14

July 2020	November 2020	January 2021	February 2021



The example above shows a pattern of increased complaints over time. It is also important to note that this precinct covers the area between 34th Street and 42nd Street which is a major tourist hotspot. This could indicate once again that as things get back to normal after the pandemic and as tourism rises there is the risk that complaints will rise as well. This type of visualization helps show changes in trends at a higher level than the other visualizations that focus more on the coordinate level and helps assess which precincts need additional resources to combat increases in complaints.

Top 10 Locations of Offenses Using PageRank

Implementation

For the PageRank analysis, points were calculated using the PageRank algorithm generated on a button click event, which takes the selected offenses and the number of days back to search for complaints to determine the most influential Location nodes. The method "page_rank_locations" is executed in the neo4j_integration.ipynb notebook. The details of the algorithm executed are described in the **PageRank Centrality Algorithm** section**.** Once the PageRank locations are determined, the following query is executed to get the nearby locations.

```
nearby_locs_query = 'match
(l1:Location)-[ln:LOCATED_NEARBY]-(l2:Location)<-[:LOCATED_AT]-(c:Complaint)-[:COMMITTED_OFFENSE]->(o:
Offense) \n\
      where l1.location_id in '+ str(list(page_rank_df.location_id.unique())) + 'and c.complaint_date >= ' + from_date + '
and o.offense in '+ str(offenses) + ' with \
      apoc.convert.toFloat(l1.latitude) as l1_lat,apoc.convert.toFloat(l1.longitude) as l1_lon,
apoc.convert.toFloat(l2.latitude) as l2_lat,apoc.convert.toFloat(l2.longitude) as l2_lon \
      return distinct [l1_lat,l1_lon] as l1,[ l2_lat,l2_lon] as l2'
```

The query will find the nearby locations where the same offenses occurred during the same time frame. The PageRank and nearby locations are sent to separate functions to draw a Marker map object and add it as a layer to the map. The "add_nearby_marker" function draws a Polyline object as well to show the connectivity between the PageRank locations to the other locations.

```
def add_marker(x):
    coord = (x['latitude'], x['longitude'])

    icon1 = AwesomeIcon(
        name='bullseye',
        marker_color='red',
        icon_color='black',
        spin=False)

    marker = Marker(icon=icon1,location=coord, title='Location: ' + str(coord)+'\n'+'Page Rank: ' +str(x['score']), draggable=False)
    page_rank_map.add_layer(marker)
```

```
def add_nearby_marker(x):
    coord = tuple(x['l2'])

    icon1 = AwesomeIcon(
        name='map-pin',
        marker_color='lightblue',
        icon_color='black',
        spin=False)

    marker = Marker(icon=icon1, location=coord, draggable=False)

    line = Polyline(locations=[x['l1'],x['l2']],
    color="white",
    fill=False)

    page_rank_map.add_layer(line)
    page_rank_map.add_layer(marker)
```

Value Added

In the example below the user selected the Robbery offense going 90 days back. The red markers are the top 10 central locations selected by the Page Rank algorithm and the blue markers are locations nearby that had the same offense occur within the same time frame.

One of the areas the Page Rank algorithm chose was in midtown Manhattan where 3 of the 10 Page Rank assigned locations were close to one another. This example shows why these locations were chosen given how central they are to the large number of other nearby locations where the same offenses occurred. By knowing the central locations you can better target your policing strategies. An example could be a location with a large volume of complaints but on the outskirts of where most of the police patrol is currently occurring. Leadership can take this data and realign their patrols to match where the central locations of complaints are happening.

Robbery 90 Days Back

Complaints by Precinct In the Past Number of Days

Implementation

The following visualization was developed in Bloom using search phrases with Cypher queries. This allows precincts to filter data only relevant to their precinct and get a view of the number of recent complaints that are happening and the offenses they are related to. It is designed to filter complaints by a user specified number of days back from the latest date on record in the database. So if the user just wants to look a week back they can just enter 7, a month back 30.

Search Phrase: Complaints in $precinct in the last $number days

Cypher Query:

```
MATCH (c:Complaint)
with date(max(c.complaint_date)) - duration('P' + $number + 'D') as past_x
with min(past_x) as min_date
MATCH
(o:GeneralOffense)<-[:COMMITTED_OFFENSE]-(c:Complaint)<-[:REPORTED]-(p:PrecinctNam
e)
where date(c.complaint_date) >= min_date and p.precinct = $precinct
return c, p, o
```

Parameters:

$precinct - PrecinctName({precinct})
$number - String

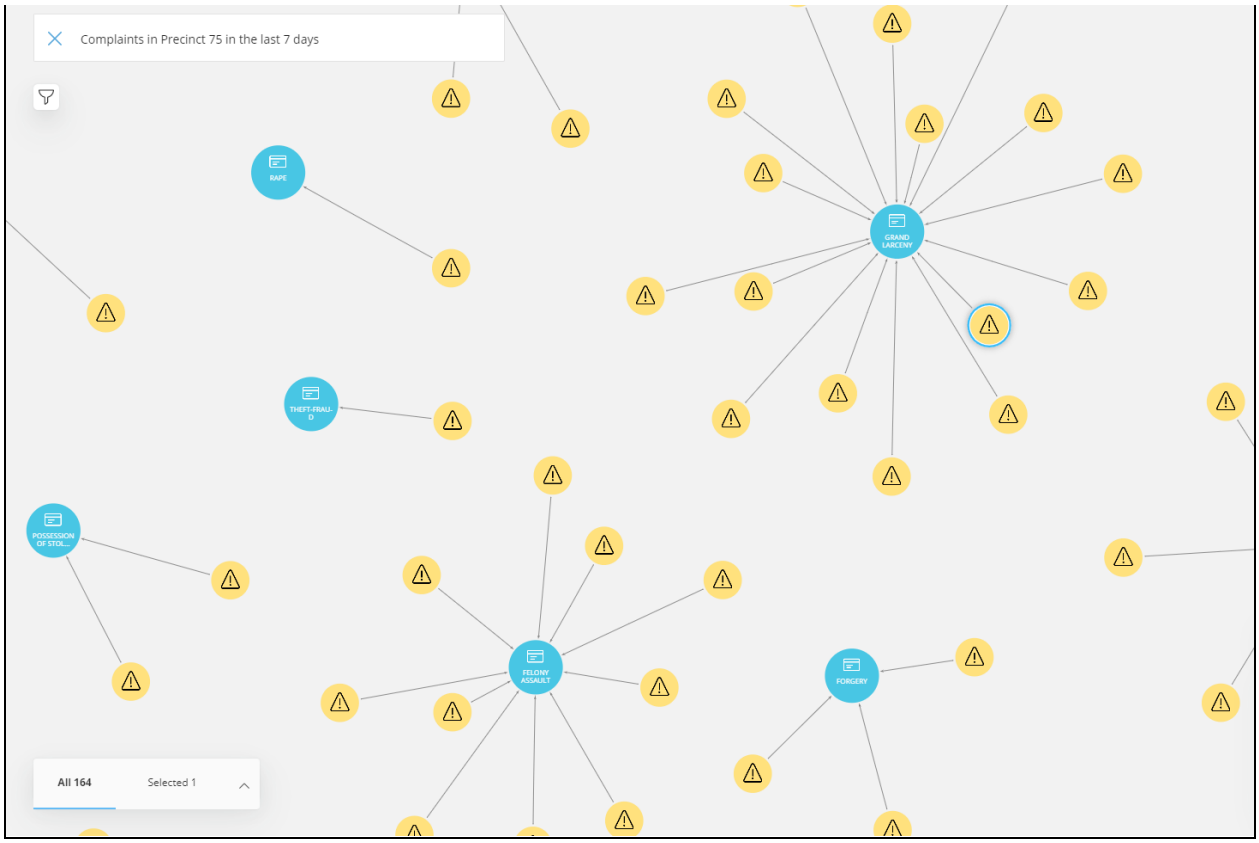
Value Added

This is more focused on what is currently happening and not for analyzing historical data months or years back. The output returns the complaint, precinct and offense nodes. The user can identify which offenses have a high degree of complaints in the selected time frame. This identifies what type of offenses each precinct should address based on the volume of complaints.

Example:

Complaints in Precinct 75 in the last 7 days

By looking at the related visualization we can see which offenses (GRAND LARCENY and FELONY ASSAULT) stand out as having a higher degree by looking at their relationship between complaints. This aims to answer the question about what is happening in each precinct.



Locations of Complaints by Precinct Within a Certain Date Range

Implementation

The following visualization was developed in Bloom using search phrases with Cypher queries. This allows precincts to filter data only relevant to their precinct and explore the relationship between complaints and locations. The user can provide a from and to date range as part of the search phrase to filter complaints only in the relevant time period they are interested in.

Search Phrase: Locations of Complaints Reported by $precinct from $complaint_from_date to $complaint_to_date

Cypher Query:

```
Match (p:PrecinctName)-[:REPORTED]->(c:Complaint)-[:LOCATED_AT]->(l1:Location)
where p.precinct = $precinct and c.complaint_date >= $complaint_from_date and
c.complaint_date <= $complaint_to_date
return  l1,c
```

Parameters:

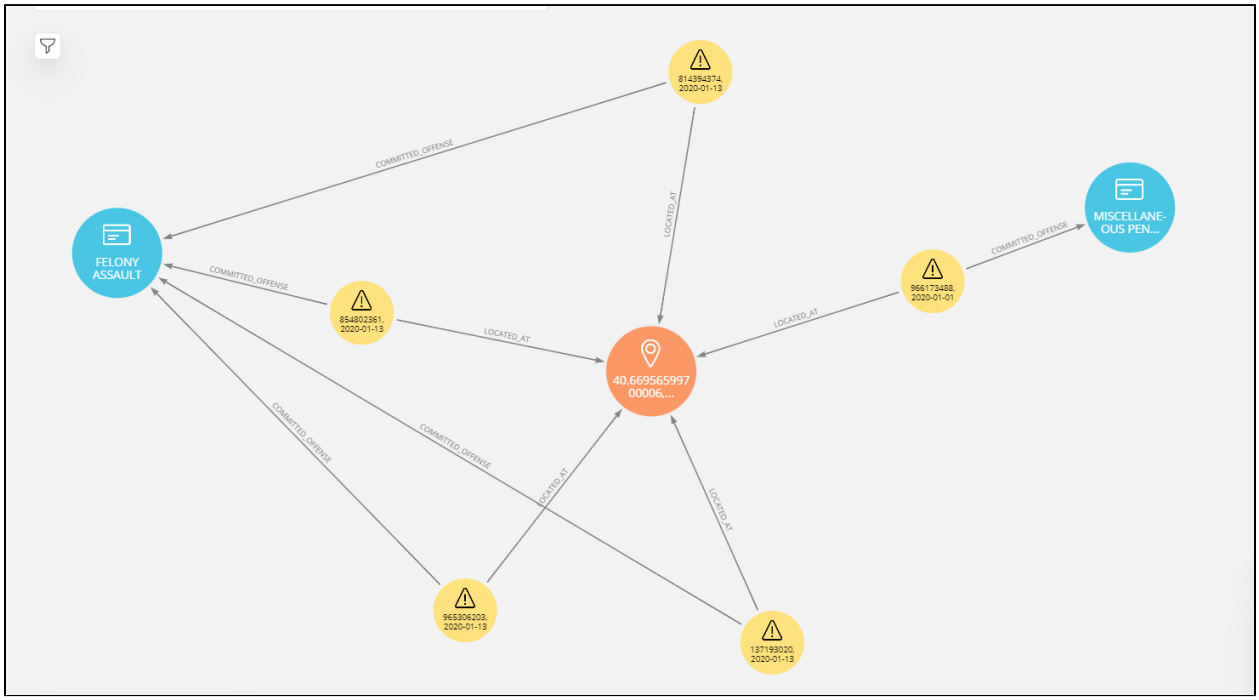
$precinct - PrecinctName({precinct})
$complaint_from_date - Complaint({complaint_date])
$complaint_to_date - Complaint({complaint_date])

Value Added

This allows the user to find patterns where a high degree of complaints are happening at the same location. The user can also expand upon the locations they find and bring in the offenses to see if the high volume of complaints involve the same offense multiple times or if there are a variety of offenses being committed at the same location. This aims to answer the question about what offenses each precinct is dealing with.

Example:
Locations of Complaints Reported by Precinct 75 from 2020-01-01 to 2020-01-14

In this example we are exploring Precinct 75 which has the most complaints in the dataset, and examine a two week time period in 2020. The relationships between Complaint and Location nodes are shown and the user can expand the relationships between the two nodes and see which locations had numerous complaints within the short window of time. The example below shows one location had 5 complaints reported. Offenses were expanded to find that 4 out of the 5 complaints involved a felony assault and those 4 complaints were reported on the same day, 1/13/2020. This aims to answer the question about where complaints are happening in each precinct.



Link Prediction of Locations

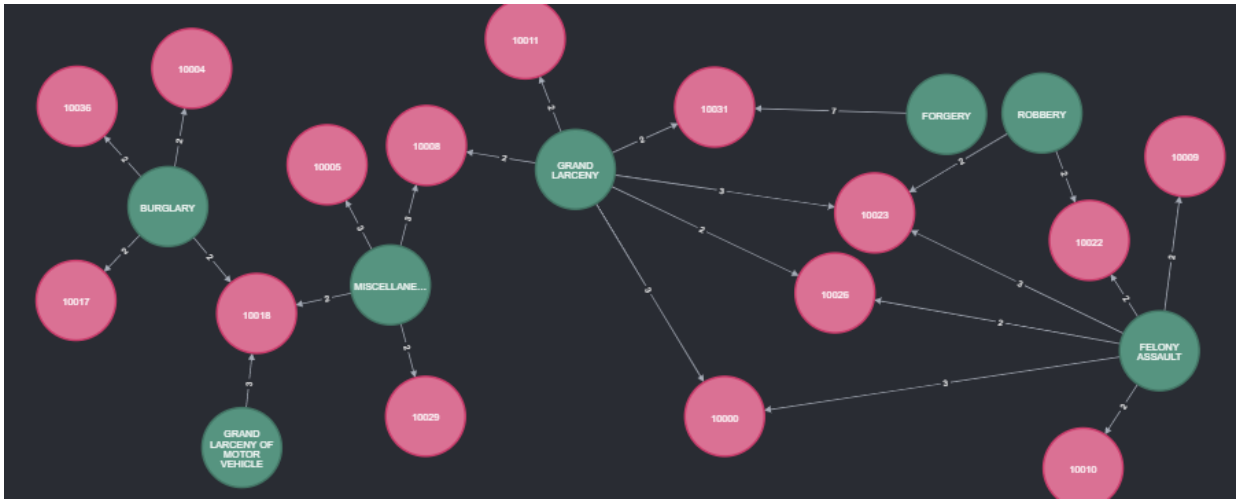
While it is important to view historical data and get a sense of what is happening in NYC in terms of complaints, this information can be used to make predictions about the future. Link prediction is a component of graph machine learning that is used to predict future relationships or find missing links between nodes. A use case for this algorithm is predicting the locations where future offenses will occur. The link prediction process and results can be found on the link_prediction.ipynb Jupyter notebook.

Since there were so many complaints and locations, a projection from Offense to Location was created. Using the query below the relationship MULTIPLE_COMPLAINTS is created, which is a relationship between only offenses to locations with more than one complaint. The relationship property 'first_complaint' indicates the earliest a complaint of that type of offense was committed at that location and 'previous_complaints' which is the amount of previous complaints.

```
match (o:Offense)<-[:COMMITTED_OFFENSE]-(c:Complaint)-[:LOCATED_AT]->(l:Location)
with c, l, o
order by l.location_id, c.complaint_date
with l,o, collect(date(c.complaint_date))[0] as date, count(*) as prev_complaints
MERGE (o)-[pc:MULTIPLE_COMPLAINTS {first_complaint: date}]->(l)
set pc.previous_complaints = prev_complaints
```

The following query was run afterwards to delete relationships that did not have multiple complaints at that location.

```
MATCH ()-[r:MULTIPLE_COMPLAINTS]->()
where r.previous_complaints <= 1
delete r
```



A graph is created that uses the Location and Offense nodes and MULTIPLE_COMPLAINTS relationship projection that is undirected.

```
call gds.graph.create('loc_offenses',
   ['Location','Offense'],
   {
      MULTIPLE_COMPLAINTS: {
         orientation: "UNDIRECTED"
            }
      }
)
```

The Neo4J splitRelationships function uses the graph from the previous step to split the data into test and training sets. The first query will create the test data by creating a holdout set using 10% of the data from the graph. The remaining data from the graph is stored in an OUTER_TRAIN relationship type. This data will be made up of both relationships and non-existing relationships (negative samples) so that the algorithm has less bias towards relationships that already exist.

```
call gds.alpha.ml.splitRelationships.mutate('loc_offenses',{
   remainingRelationshipType: 'OUTER_TRAIN',
   holdoutRelationshipType: 'TEST',
   holdoutFraction: 0.1,
   negativeSamplingRatio: 1.0,
   randomSeed: 1
})
```

The splitRelationships function is called once again to use the data in the OUTER_TRAIN relationship created in the previous step in order to create the training data. The remaining data is used for the embedding algorithm.

```
call gds.alpha.ml.splitRelationships.mutate('loc_offenses',{
   relationshipTypes: ['OUTER_TRAIN'],
   remainingRelationshipType: 'EMBEDDING_GRAPH',
   holdoutRelationshipType: 'TRAIN',
   holdoutFraction: 0.1,
   negativeSamplingRatio: 1.0,
   randomSeed: 1
})
```

Fast Random Projection was used as the embedding algorithm, which first creates a sparse projection randomly and then iteratively improves and creates a vector for each node. The data used is from the EMBEDDING_GRAPH data created in the previous step. A property called 'frp' is created with the embedding results.

```
call gds.beta.fastRPExtended.mutate('loc_offenses', {
   relationshipTypes: ['EMBEDDING_GRAPH'],
   mutateProperty: 'frp',
   embeddingDimension: 512,
```

```
  propertyDimension: 256,
  randomSeed: 1
})
```

The model will then get trained using the feature property 'frp' created by the embedding algorithm in the previous step. It will use cross fold validation to sample data and divide it into the validationFolds specified where each group will be the holdout set once. The model will train on different model parameters such as the penalty, epochs and the link feature combiner, the best performing parameters are chosen from the provided params property. Both the Area Under the Precision-Recall Curve for the test and training data are returned along with the model parameters chosen.

```
call gds.alpha.ml.linkPrediction.train('loc_offenses',{
  trainRelationshipType: 'TRAIN',
  testRelationshipType: 'TEST',
  modelName: 'model',
  featureProperties: ['frp'],
  validationFolds: 5,
  negativeClassWeight: 1.0,
  randomSeed: 1,
  concurrency: 4,
  params: [
    {penalty: 0.0001, maxEpochs: 500, linkFeatureCombiner: 'HADAMARD'},
    {penalty: 1.0, maxEpochs: 500, linkFeatureCombiner: 'HADAMARD'},
    {penalty: 10000.0, maxEpochs: 500, linkFeatureCombiner: 'HADAMARD'},
    {penalty: 0.0001, maxEpochs: 500, linkFeatureCombiner: 'L2'},
    {penalty: 1.0, maxEpochs: 500, linkFeatureCombiner: 'L2'},
    {penalty: 10000.0, maxEpochs: 500, linkFeatureCombiner: 'L2'},
    {penalty: 0.0001, maxEpochs: 500, linkFeatureCombiner: 'COSINE'},
    {penalty: 1.0, maxEpochs: 500, linkFeatureCombiner: 'COSINE'},
    {penalty: 10000.0, maxEpochs: 500, linkFeatureCombiner: 'COSINE'}
  ]
})
yield modelInfo
return modelInfo.metrics.AUCPR.test as test_auc, modelInfo.metrics.AUCPR.outerTrain as train_auc,
modelInfo.bestParameters as best_parameters
```

The results show a test AUCPR of .940065 and a training AUCPR of .948035 and the best model chosen was the L2 link feature combiner with a .0001 penalty.

test_auc	train_auc	best_parameters

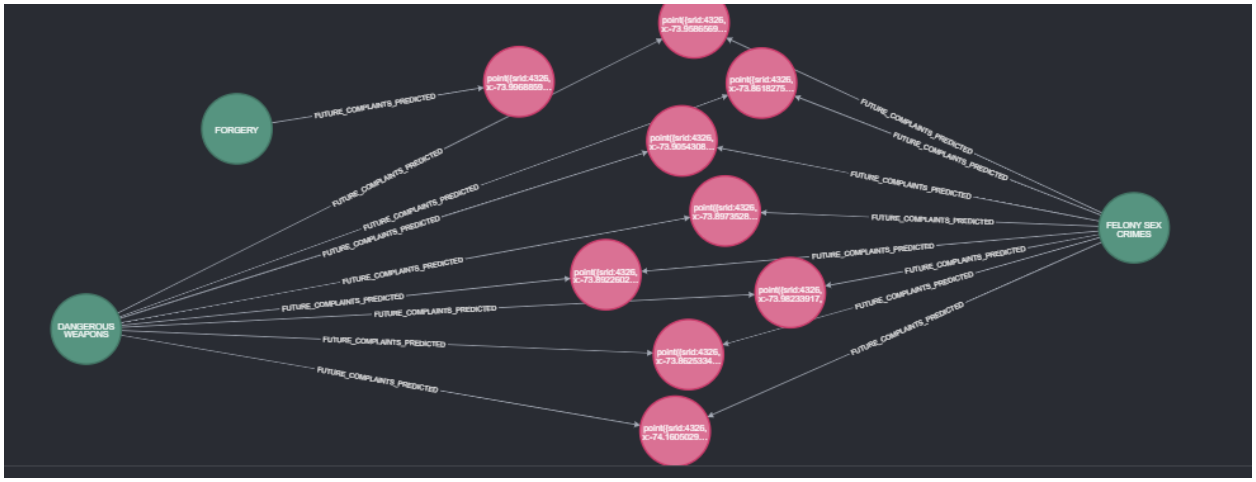
0.9400651747704089	0.9480345643530949	{
		"maxEpochs": 500,
		"minEpochs": 1,
		"penalty": 0.0001,
		"patience": 1,
		"linkFeatureCombiner": "L2",
		"batchSize": 100,
		"sharedUpdater": false,
		"tolerance": 0.001,
		"concurrency": 4
		}

The prediction method is then called to make predictions on future relationships using the trained model from the previous step. A new relationship created is called FUTURE_COMPLAINTS_PREDICTED. It includes a relationship property 'probability' containing the probability of the prediction. For the purpose of time, only the top 50 results were generated and a threshold of .3 was used to filter probabilities. This is a very intensive step as it runs the entire prediction across the entire graph. Using the prediction method below took over 48 minutes to complete.

```
CALL gds.alpha.ml.linkPrediction.predict.write('loc_offenses', {
  relationshipTypes: ['MULTIPLE_COMPLAINTS'],
  modelName: 'loc_offenses_model',
  writeRelationshipType: 'FUTURE_COMPLAINTS_PREDICTED',
  topN: 50,
  threshold: 0.3
}) YIELD relationshipsWritten
```

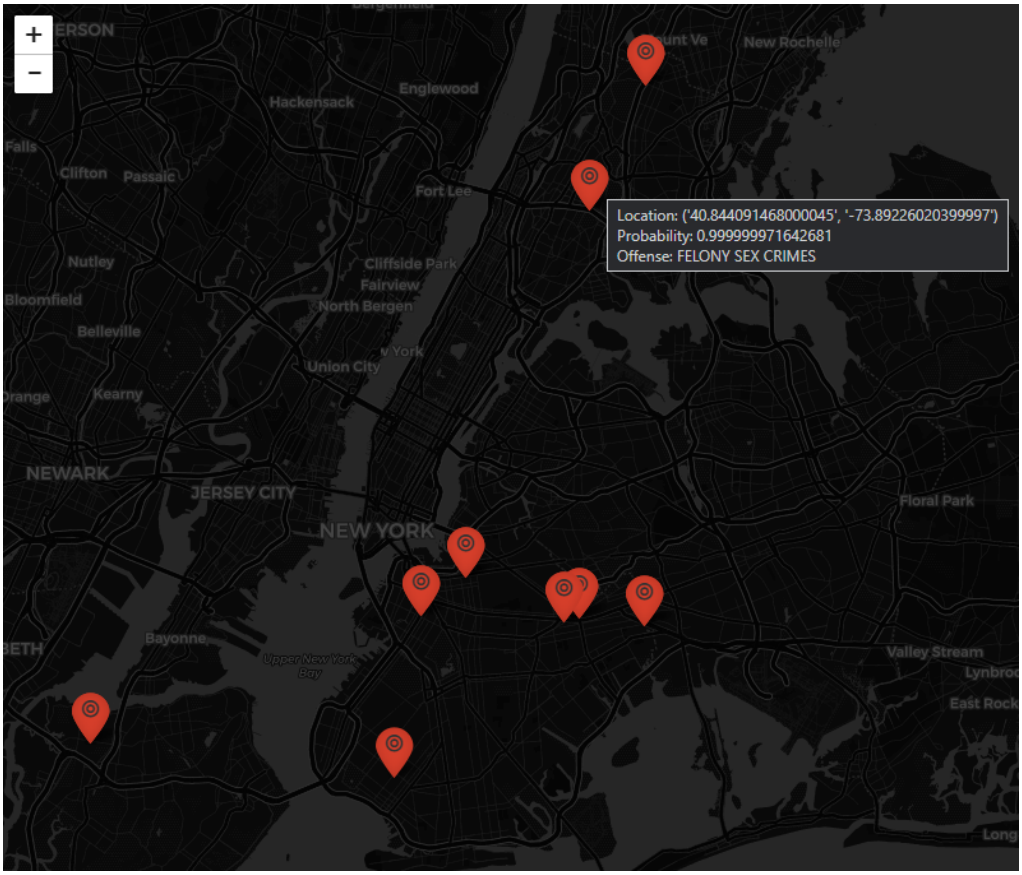
Once complete the new relationship is created. Some relationships are monopartite projections between the Location nodes. For the purposes of this analysis the following query was run to get which offenses had a high probability of occurring at a location.

```
match p = (o:Offense)-[:FUTURE_COMPLAINTS_PREDICTED]->(l:Location)
return p
```



Three offenses are listed, Forgery, Dangerous Weapons and Felony Sex Crimes. While forgery is predicted to occur at one location, the remaining locations are common to both offenses. This could potentially indicate high crime areas, more analysis would be needed to run the prediction algorithm with more results other than the top 50.

Below is a list of where these predicted locations are on a map and you can hover over the icon to see the probability and the offense being predicted.



Conclusion & Value Added

In conclusion there were three questions to answer, what is happening in terms of the offenses being committed in addition to when and where these offenses are occurring in high volumes. The visualization tools created give the NYPD the ability to examine these questions at different levels of detail and the flexibility to analyze different combinations of offenses and date ranges..

Users can compare precincts during different periods of time to identify if the complaint level is rising, declining or staying the same. Diving deeper the user can look at where hotspots of offenses are occurring within a specified date range through the use of a heatmap visualization and Louvain community detection algorithm. Finally, at the coordinate level the PageRank centrality algorithm finds which location nodes are the most influential in the dataset based on the offenses and date range selected. It also accounts for nearby locations where similar offenses have occurred.

The NYPD can determine where more resources need to be allocated for a precinct to combat a certain type of offense. Hotspots can identify what sections of NYC are being affected and if there are any geographical patterns such as if only a certain borough is being affected by the offense. Lastly, central locations of offenses can lead to adding temporary police locations, realigning patrols or adding cameras or other technology to help reduce the complaint rates.

WIth the flexibility to analyze different time periods, different periods of the pandemic can also be studied. As an example, it showed that grand larceny was very low during the time of the pandemic but normally it is a leading offense in terms of the number of complaints. In the unfortunate event that something such as a pandemic would occur again the NYPD would have knowledge about what offenses are higher or lower than normal.

By having the data stored in a graph database, another advantage is using Neo4J's new machine learning library to make predictions. Link prediction was used to try and predict the probability of complaints occurring at a specific location in the future. This information would be very valuable to the NYPD, because it could help them determine different policing strategies based on where these complaints are predicted to be occurring in the future. Such strategies could be increasing patrols in these areas or outreach with community leaders in at-risk areas. Having link prediction implemented and other machine learning models can help the NYPD stay ahead of the trends in complaints and it gives them an edge in the fight to prevent rising complaints.

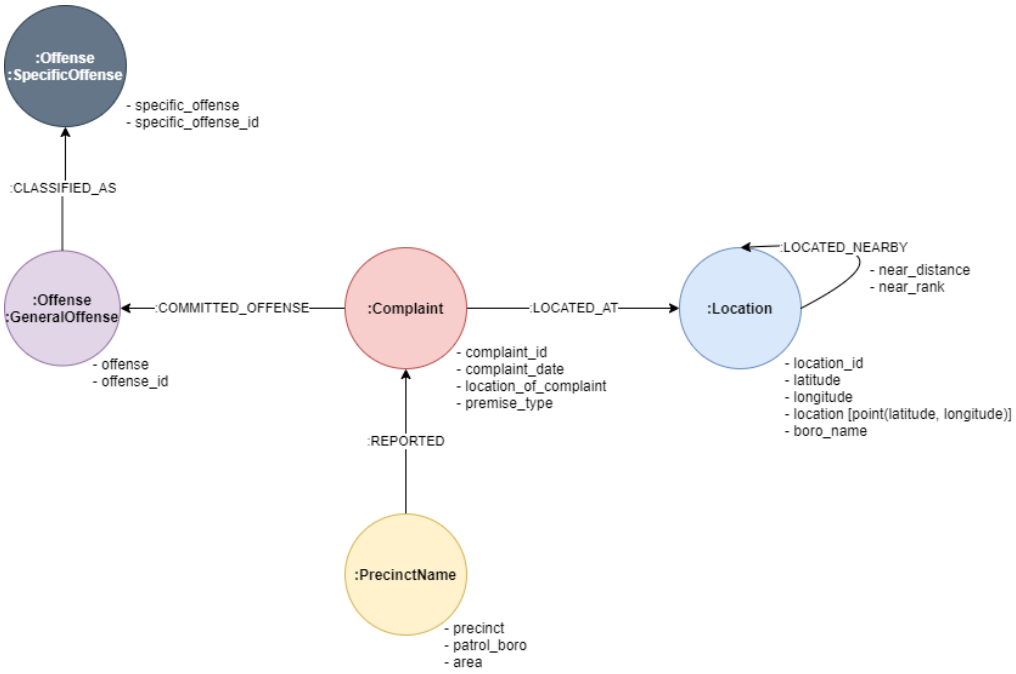
In conclusion, the goal was to provide tools to the NYPD to be able spot trends in complaints. This was accomplished by using different tools in Neo4J. The first step was querying the data that was loaded into the database to get some initial insight on what type of offenses are the highest and where most complaints occur. It then evolved deeper into using algorithms to find clusters of similar offenses and which locations are central to these offenses. The visualizations allow the NYPD to examine this data with the flexibility of choosing different time periods and different offenses and developing these visualizations in the form of a map was critical to be

able to show where these patterns are occurring. Using this historical data, a story can be told about how complaints affect an area over time or the effect a pandemic has on certain offenses and leveraging this data to not just tell what has happened but predicting what will happen.

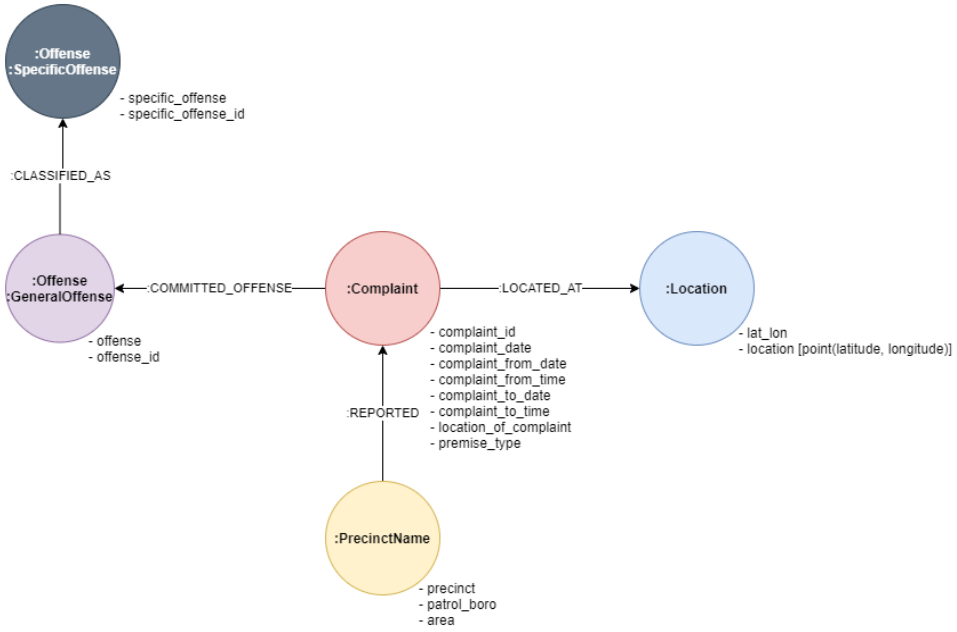
Appendix

Appendix A - Graph Data Model Revisions

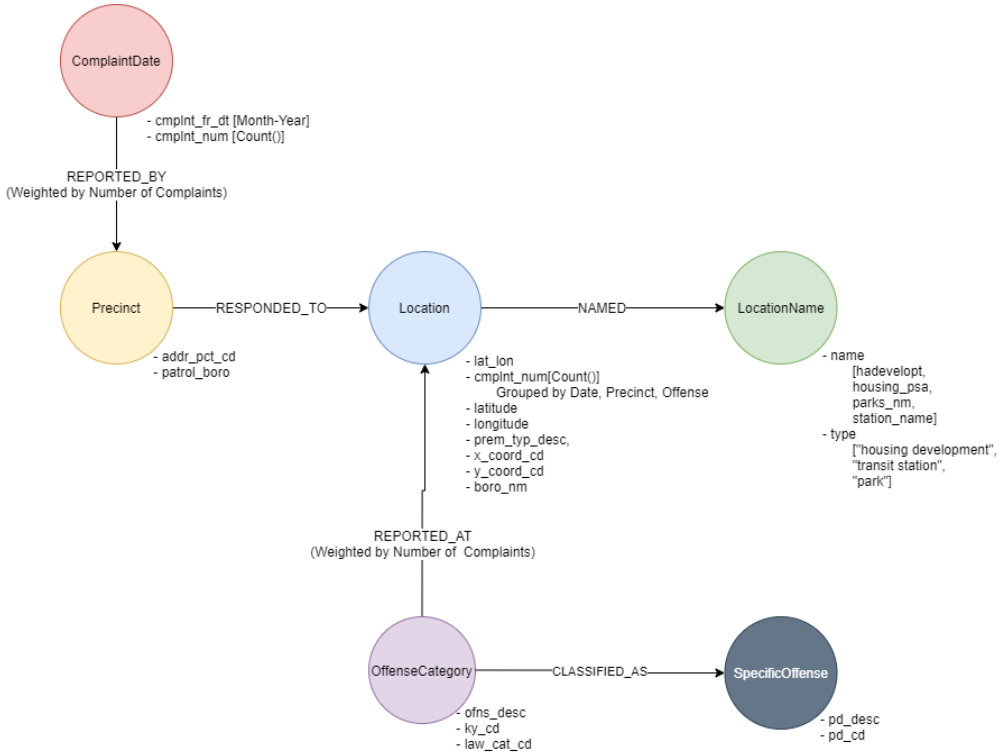
Revision 2 Data Model



Revision 1 Data Model



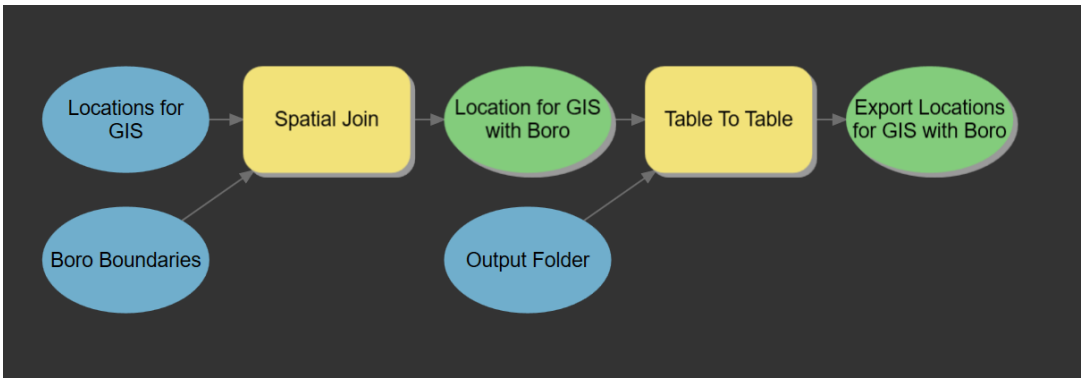
Original Data Model

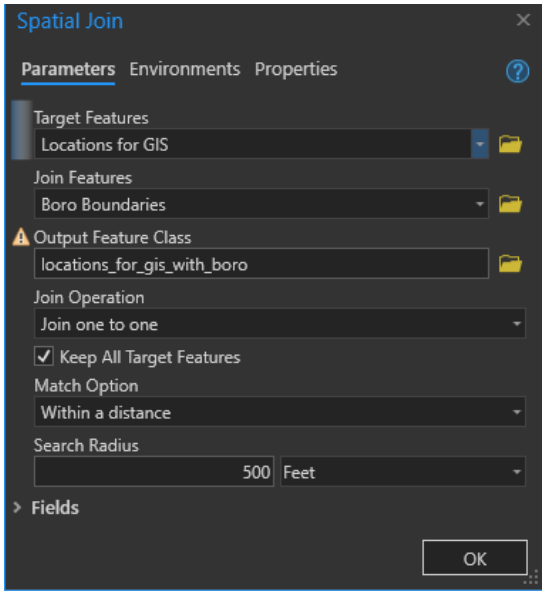


Appendix B - GIS Analysis

Since the data set had latitude and longitude, there were certain tasks that could take advantage of GIS tools to extract information from spatial data. The first analysis was getting the boro associated with each location. The locations were brought in as points and added to a map and projected to a state plane coordinate system.

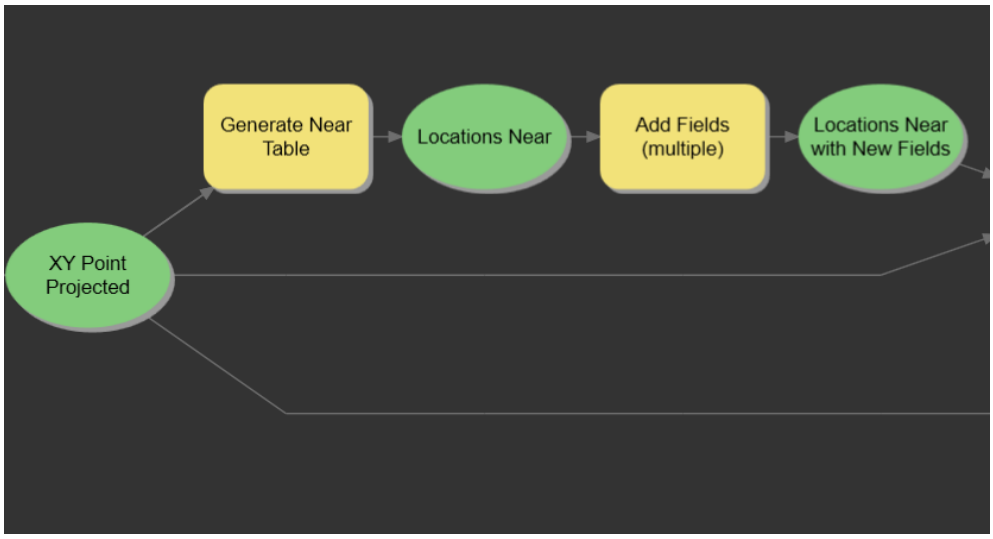
A model was created that performs what is called a Spatial Join and will join the points with the selected boro attribute if the points are within a 500ft radius of the boro boundaries polygon from NYC Open Data. All points will then have the boro joined with the data and the joined data is then exported to a CSV and brought back into the Jupyter notebook.

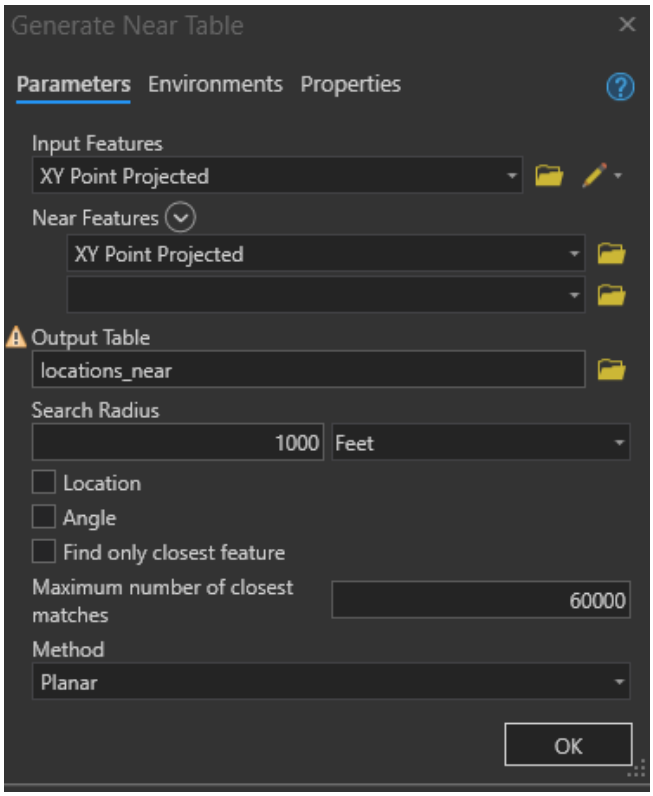




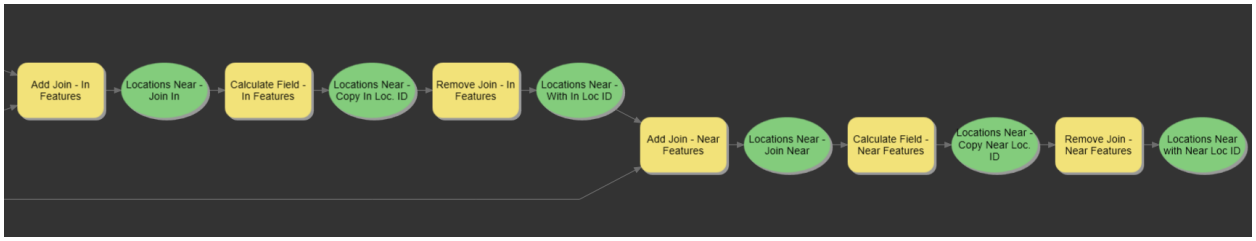
The second analysis performed calculated the distance between locations within a 1000ft. The first part was similar to the previous analysis described, where the data is brought in as points and projected to the state plane feet coordinate system. The Generate Near Table was used to calculate a distance and ranking between the locations. In the example below both the input and

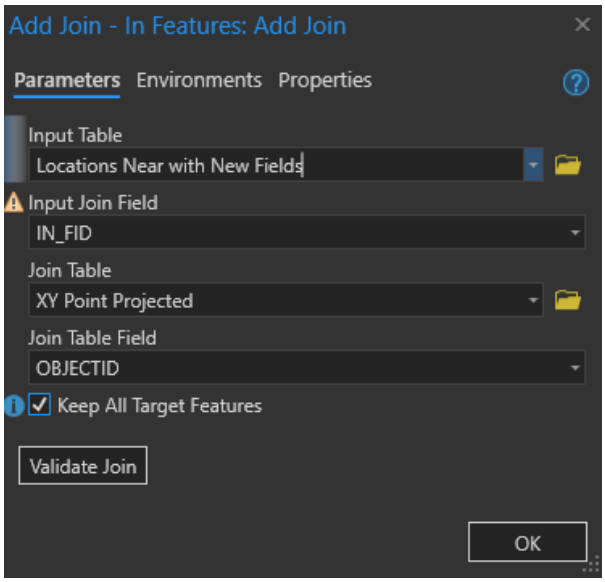
near features point to the same projected locations and a 1000ft search radius is specified as well as a large maximum number of closest features to return, to ensure we get all the nearby locations returned.

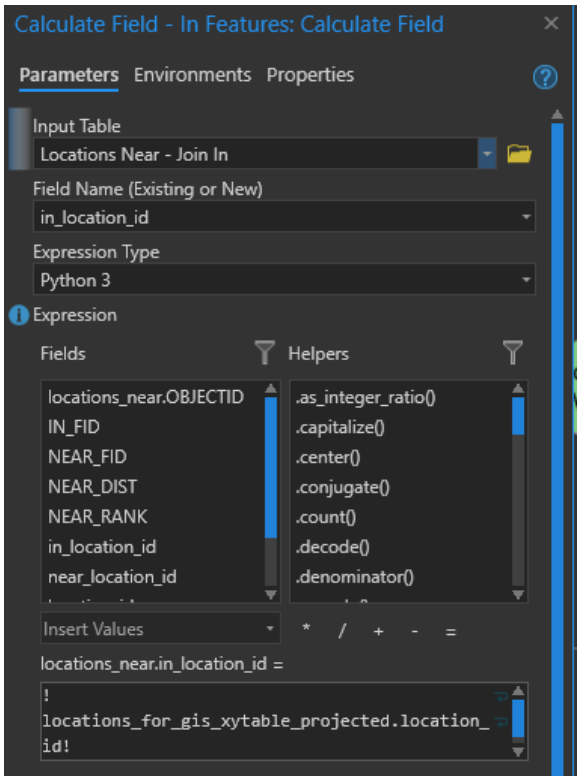




The results are returned using not the location id we created but with a software standard field called OBJECTID. The remaining parts of the model are joining the near table results with the projected points by OBJECTID and capturing a location id in two fields, “in_location_id” and “near_location_id”. The results can then be exported to a CSV and used to load into Neo4J for use in the analysis.







OBJECTID	IN_FID	NEAR_FID	NEAR_DIST	NEAR_RANK	in_location_id	near_location_id
1	1	4	473.5236354	1	0	3
2	1	8	571.9479985	2	0	7
3	1	7	629.578843	3	0	6
4	1	6	774.9730987	4	0	5
5	1	11	853.8673133	5	0	10
6	1	5	969.7113093	6	0	4
7	2	3	501.8177273	1	1	2
8	2	15	971.0329815	2	1	14