Nearest Neighbors Lab

Introduction

In this lab, you apply nearest neighbors technique to help a taxi company predict the length of their rides. Imagine that we are hired to consult for LiftOff, a limo and taxi service that is just opening up in NYC. Liftoff wants it's taxi drivers to target longer rides, as the longer the ride the more money it makes. LiftOff has the following theory:

- the pickup location of a taxi ride can help predict the length of the ride.

LiftOff asks us to do some analysis to write a function that will allow it to **predict the length of a taxi ride for any given location **.

Our technique will be the following:

- **Collect** Obtain the data containing all of the taxi information, and only select the attributes of taxi trips that we need
- ** Explore ** Examine the attributes of our data, and plot some of our data on a map
- ** Train ** Write our nearest neighbors formula, and change the number of nearby trips to predict the length of a new trip
- ** Predict ** Use our function to predict trip lengths of new locations

Collect and Explore the data

Collect the Data

Luckily for us, [NYC Open Data (https://opendata.cityofnewyork.us/)](https://opendata.cityofnewyork.us/) collects information about NYC taxi trips and provides this data on [its website (https://data.cityofnewyork.us/Transportation/2014-Yellow-Taxi-Trip-Data/gn7m-em8n)](https://data.cityofnewyork.us/Transportation/2014-Yellow-Taxi-Trip-Data/gn7m-em8n).

NYC OpenData Home Data

2014 Yellow Taxi Trip Data

Filter cards by selecting date ranges on date/time cards, typing freeform text into search cards, or clicking the columns in the column card.

This dataset includes trip records from all trips completed in yellow taxis in NYC in 2014. Records include fields capturing pick-up and drop-off dates/times, pick-up and drop-off locations, trip distances, itemized fares, rate types, payment types, and driver-reported passenger counts. The data used in the attached datasets were collected and provided to the NYC Taxi and Limousine Commission (TLC) by technology providers authorized under the Taxicab Passenger Enhancement Program (TPEP). The trip data was not created by the TLC, and TLC makes no representations as to the accuracy of these data.
Data Dictionary for this dataset can be found here:
http://www.nyc.gov/html/tlc/downloads/pdf/data_dictionary_trip_records_yellow.pdf

Show less

Export API

For your reading pleasure, the data has already been downloaded into the trips.json (https://github.com/learn-co-curriculum/nearest-neighbors-lab/blob/master/trips.json) file in this lab which you can find here. We'll use Python's `json` library to take the data from the `trips.json` file and store it as a variable in our notebook.

In [1]:
```
import json
# First, read the file
trips_file = open('trips.json')
# Then, convert contents to list of dictionaries
trips = json.load(trips_file)
```

> Press shift + enter

Explore the data

The next step is to explore the data. First, let's see how many trips we have.

In [2]:
```
len(trips)
```

Out[2]: 1000

Not bad at all. Now let's see what each individual trip looks like. Each trip is a dictionary, so we can see the attributes of each trip with the `keys` function.

In [3]:
```
print(trips[0].keys())
print("*** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***
print(trips[0].values())
print("*** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***
print(trips[0].items())
```

```
dict_keys(['dropoff_datetime', 'dropoff_latitude', 'dropoff_longitude',
'fare_amount', 'imp_surcharge', 'mta_tax', 'passenger_count', 'payment_ty
pe', 'pickup_datetime', 'pickup_latitude', 'pickup_longitude', 'rate_cod
e', 'tip_amount', 'tolls_amount', 'total_amount', 'trip_distance', 'vendo
r_id'])
*** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *
** *** *** *** *** *** *
dict_values(['2014-11-26T22:31:00.000', '40.746769999999998', '-73.997450
000000001', '52', '0', '0.5', '1', 'CSH', '2014-11-26T21:59:00.000', '40.
64499', '-73.781149999999997', '2', '0', '5.3300000000000001', '57.829999
999999998', '18.379999999999999', 'VTS'])
*** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *
** *** *** *** *** *** *
dict_items([('dropoff_datetime', '2014-11-26T22:31:00.000'), ('dropoff_la
titude', '40.746769999999998'), ('dropoff_longitude', '-73.99745000000000
1'), ('fare_amount', '52'), ('imp_surcharge', '0'), ('mta_tax', '0.5'),
('passenger_count', '1'), ('payment_type', 'CSH'), ('pickup_datetime', '2
014-11-26T21:59:00.000'), ('pickup_latitude', '40.64499'), ('pickup_longi
tude', '-73.781149999999997'), ('rate_code', '2'), ('tip_amount', '0'),
('tolls_amount', '5.3300000000000001'), ('total_amount', '57.829999999999
998'), ('trip_distance', '18.379999999999999'), ('vendor_id', 'VTS')])
```

Limit our data

Ok, now that we have explored some of our data, let's begin to think through what data is relevant for our task.

Remember that our task is to **use the trip location to predict the length of a trip**. So let's select the `pickup_latitude` , `pickup_longitude` , and `trip_distance` from each trip. That will give us the trip location and related `trip_distance` for each trip. Then based on these **actual** trip distances we can use nearest neighbors to predict an **expected** trip distance for a trip, provided an **actual** location.

** Add in about trip distance **

Write a function called `parse_trips(trips)` that returns a list of the trips with only the following attributes:

- `trip_distance`
- `pickup_latitude`
- `pickup_longitude`

In [4]:
```
def parse_trips(trips):
    trips_clone = trips.copy()
    A = []
    for i in range(0,len(trips_clone)):
        X = {}
        X = {k:v for k,v in trips_clone[i].items() if k == 'trip_distance'
        #print(X)
        A.append(X)
    return A
    pass

parse_trips(trips)
#trips_clone
#for trip in trips_clone:
#print("distance:"+str(trip['trip_distance'])+", latitude:"+str(trip['picku
#print("*** *** *** *** *** ***")
#for k,v in trip.items():
#if (k != 'trip_distance' and k != 'pickup_latitude' and k != 'pickup_longi
#del trip[k]
#print(k+":"+str(v))
#A.setdefault(k, []).append(v)
#A
```

Out[4]:
```
[{'pickup_latitude': '40.64499',
  'pickup_longitude': '-73.78114999999997',
  'trip_distance': '18.379999999999999'},
 {'pickup_latitude': '40.766931',
  'pickup_longitude': '-73.982097999999993',
  'trip_distance': '1.3'},
 {'pickup_latitude': '40.777729999999998',
  'pickup_longitude': '-73.951902000000004',
  'trip_distance': '4.5'},
 {'pickup_latitude': '40.795678000000002',
  'pickup_longitude': '-73.971048999999994',
  'trip_distance': '2.3999999999999999'},
 {'pickup_latitude': '40.762912',
  'pickup_longitude': '-73.967782',
  'trip_distance': '0.83999999999999997'},
 {'pickup_latitude': '40.731175999999998',
  'pickup_longitude': '-73.991572000000005',
  'trip_distance': '0.80000000000000004'},
 {'pickup_latitude': '40.800218999999998',
```

In [5]:
```
parsed_trips = parse_trips(trips)
parsed_trips and parsed_trips[0]

# {'pickup_latitude': '40.64499',
#  'pickup_longitude': '-73.78115',
#  'trip_distance': '18.38'}
```

Out[5]:
```
{'pickup_latitude': '40.64499',
 'pickup_longitude': '-73.78114999999997',
 'trip_distance': '18.379999999999999'}
```

Now, there's just one change to make. If you look at one of the trips, all of the values are strings. Let's change them to be floats.

Let's change them to be floats.

In [6]:
```python
def float_values(trips):
    parsed_trips = parse_trips(trips)
    A = []
    for i in range(0,len(parsed_trips)):
        X = {k:float(v) for k,v in parsed_trips[i].items()}
        A.append(X)

    return A
    pass

#trips_clone = trips.copy()
#A = []
#for i in range(0,len(trips_clone)):
#X = {}
#X = {k:float(v) for k,v in trips_clone[i].items() if k == 'trip_distance'
#print(X)
#A.append(X)
#return A
#pass

float_values(trips)
```

```
  'trip_distance': 18.38},
 {'pickup_latitude': 40.766931,
  'pickup_longitude': -73.982098,
  'trip_distance': 1.3},
 {'pickup_latitude': 40.77773,
  'pickup_longitude': -73.951902,
  'trip_distance': 4.5},
 {'pickup_latitude': 40.795678,
  'pickup_longitude': -73.971049,
  'trip_distance': 2.4},
 {'pickup_latitude': 40.762912,
  'pickup_longitude': -73.967782,
  'trip_distance': 0.84},
 {'pickup_latitude': 40.731176,
  'pickup_longitude': -73.991572,
  'trip_distance': 0.8},
 {'pickup_latitude': 40.800219,
  'pickup_longitude': -73.968098,
  'trip_distance': 0.5},
 {'pickup_latitude': 40.648509,
```

In [7]:
```python
cleaned_trips = float_values(parsed_trips)
```

In [8]:
```python
cleaned_trips[0]

# {'pickup_latitude': 40.64499,
#  'pickup_longitude': -73.78115,
#  'trip_distance': 18.38}
```

Out[8]:
```
{'pickup_latitude': 40.64499,
 'pickup_longitude': -73.78115,
 'trip_distance': 18.38}
```

Exploring the Data

Now that we have paired down our data, let's get a sense of our trip data. We can use the `folium` Python library to plot a map of Manhattan, and our data. First we must import `folium`, and then use the `Map` function to pass through a `location`, and `zoom_start`. If a map isn't showing up below, copy and paste the command `pip install -r requirements.txt` into your terminal to install `folium` then try again.

In [9]:
```
!pip install folium
!pip install -r requirements.txt
import folium
manhattan_map = folium.Map(location=[40.7589, -73.9851], zoom_start=11)
manhattan_map.location
```

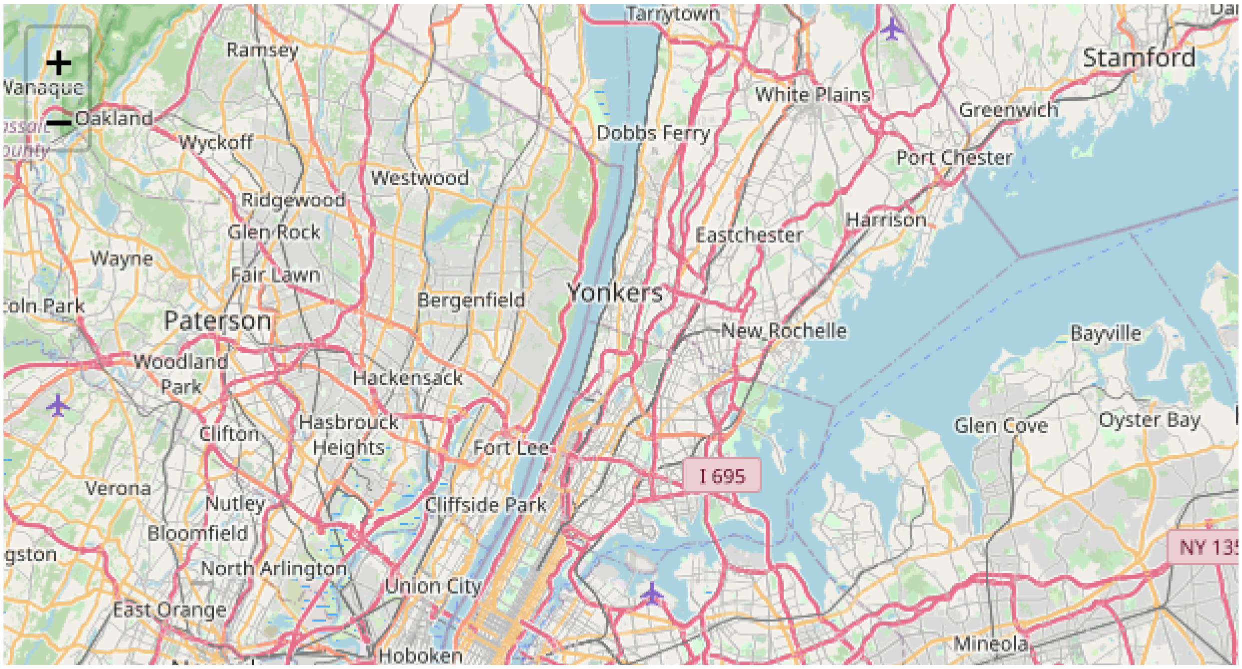
```
Requirement already satisfied: folium in /opt/conda/envs/learn-env/lib/py
thon3.6/site-packages (0.10.0)
Requirement already satisfied: numpy in /opt/conda/envs/learn-env/lib/pyt
hon3.6/site-packages (from folium) (1.17.2)
Requirement already satisfied: branca>=0.3.0 in /opt/conda/envs/learn-en
v/lib/python3.6/site-packages (from folium) (0.3.1)
Requirement already satisfied: jinja2>=2.9 in /opt/conda/envs/learn-env/l
ib/python3.6/site-packages (from folium) (2.10.3)
Requirement already satisfied: requests in /opt/conda/envs/learn-env/lib/
python3.6/site-packages (from folium) (2.22.0)
Requirement already satisfied: six in /opt/conda/envs/learn-env/lib/pytho
n3.6/site-packages (from branca>=0.3.0->folium) (1.12.0)
Requirement already satisfied: MarkupSafe>=0.23 in /opt/conda/envs/learn-
env/lib/python3.6/site-packages (from jinja2>=2.9->folium) (1.1.1)
Requirement already satisfied: certifi>=2017.4.17 in /opt/conda/envs/lear
n-env/lib/python3.6/site-packages (from requests->folium) (2019.9.11)
Requirement already satisfied: idna<2.9,>=2.5 in /opt/conda/envs/learn-en
v/lib/python3.6/site-packages (from requests->folium) (2.8)
Requirement already satisfied: chardet<3.1.0,>=3.0.2 in /opt/conda/envs/l
earn-env/lib/python3.6/site-packages (from requests->folium) (3.0.4)
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in
/opt/conda/envs/learn-env/lib/python3.6/site-packages (from requests->fol
ium) (1.24.2)
Requirement already satisfied: folium in /opt/conda/envs/learn-env/lib/py
thon3.6/site-packages (from -r requirements.txt (line 1)) (0.10.0)
Requirement already satisfied: requests in /opt/conda/envs/learn-env/lib/
python3.6/site-packages (from folium->-r requirements.txt (line 1)) (2.2
2.0)
Requirement already satisfied: branca>=0.3.0 in /opt/conda/envs/learn-en
v/lib/python3.6/site-packages (from folium->-r requirements.txt (line 1))
(0.3.1)
Requirement already satisfied: numpy in /opt/conda/envs/learn-env/lib/pyt
hon3.6/site-packages (from folium->-r requirements.txt (line 1)) (1.17.2)
Requirement already satisfied: jinja2>=2.9 in /opt/conda/envs/learn-env/l
ib/python3.6/site-packages (from folium->-r requirements.txt (line 1))
(2.10.3)
Requirement already satisfied: certifi>=2017.4.17 in /opt/conda/envs/lear
n-env/lib/python3.6/site-packages (from requests->folium->-r requirement
s.txt (line 1)) (2019.9.11)
Requirement already satisfied: chardet<3.1.0,>=3.0.2 in /opt/conda/envs/l
earn-env/lib/python3.6/site-packages (from requests->folium->-r requireme
nts.txt (line 1)) (3.0.4)
Requirement already satisfied: idna<2.9,>=2.5 in /opt/conda/envs/learn-en
v/lib/python3.6/site-packages (from requests->folium->-r requirements.txt
(line 1)) (2.8)
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in
/opt/conda/envs/learn-env/lib/python3.6/site-packages (from requests->fol
ium->-r requirements.txt (line 1)) (1.24.2)
Requirement already satisfied: six in /opt/conda/envs/learn-env/lib/pytho
n3.6/site-packages (from branca>=0.3.0->folium->-r requirements.txt (line
1)) (1.12.0)
```

```
Requirement already satisfied: MarkupSafe>=0.23 in /opt/conda/envs/learn-
env/lib/python3.6/site-packages (from jinja2>=2.9->folium->-r requirement
s.txt (line 1)) (1.1.1)
```

Out[9]:
```
[40.7589, -73.9851]
```

In [10]:
```python
manhattan_map
```

Out[10]:



Ok, now let's see how we could add a dot to mark a specific location. We'll start with Times Square.

In [11]:
```python
marker = folium.CircleMarker(location = [40.7589, -73.9851], radius=6)
marker.add_to(manhattan_map)
print("location="+str(marker.location))
print("radius="+str(marker.options['radius']))
marker.options
```

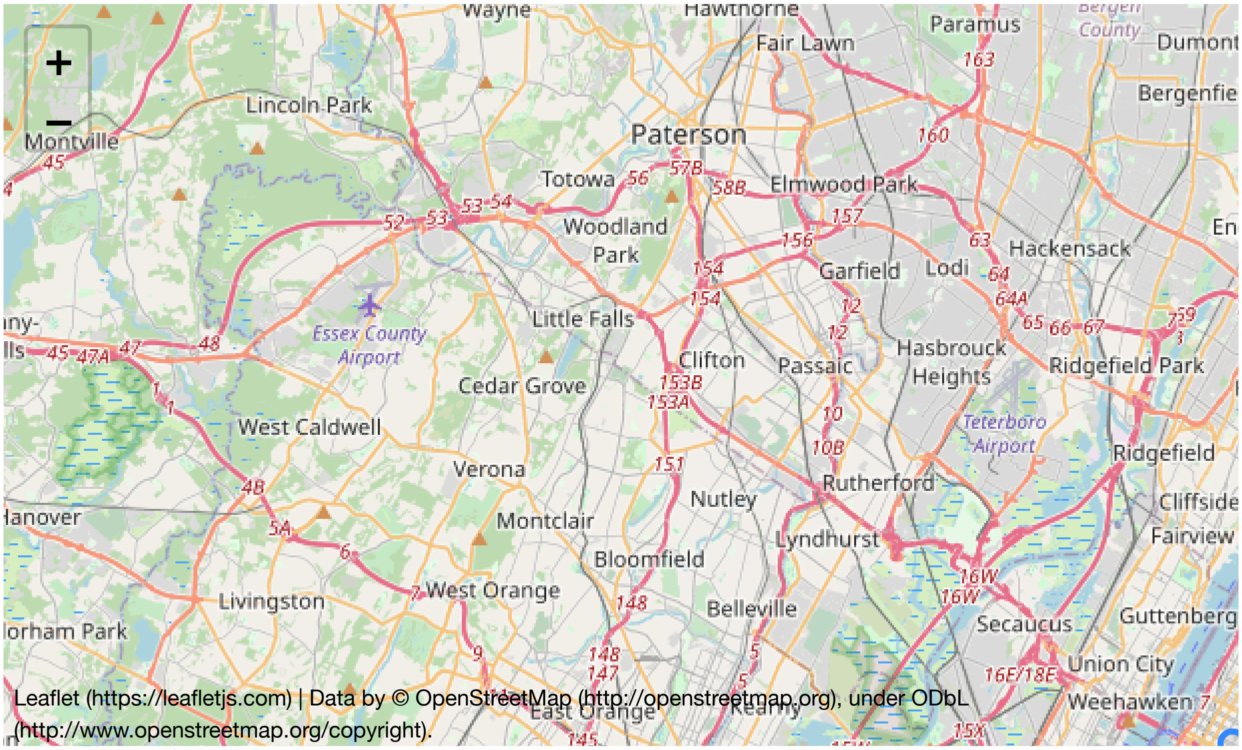
```
location=[40.7589, -73.9851]
radius=6
```

Out[11]:
```
{'stroke': True,
 'color': '#3388ff',
 'weight': 3,
 'opacity': 1.0,
 'lineCap': 'round',
 'lineJoin': 'round',
 'dashArray': None,
 'dashOffset': None,
 'fill': False,
 'fillColor': '#3388ff',
 'fillOpacity': 0.2,
 'fillRule': 'evenodd',
 'bubblingMouseEvents': True,
 'radius': 6}
```

Above, we first create a marker. Then we add that circle marker to the `manhattan_map` we created earlier.

In [12]:
```
manhattan_map
```

Out[12]:



Do you see that blue dot near Time's Square? That is our marker.

So now that we can plot one marker on a map, we should have a sense of how we can plot many markers on a map to display our taxi ride data. We simply plot a map, and then we add a marker for each location of a taxi trip.

Now let's write some functions to allow us to plot maps and add markers a little more easily.

Writing some map plotting functions

As a first step towards this, note that the functions to create both a marker and map each take in a location as two element list, representing the latitude and longitude values. Take another look:

```
marker = folium.CircleMarker(location = [40.7589, -73.9851])
manhattan_map = folium.Map(location=[40.7589, -73.9851])
```

So let's write a function called to create this two element list from a trip. Write a function called `location` that takes in a trip as an argument and returns a list where the first element is the latitude and the second is the longitude. Remember that a location looks like the following:

In [13]:
```
first_trip = {'pickup_latitude': 40.64499, 'pickup_longitude': -73.78115,
first_trip
```

Out[13]:
```
{'pickup_latitude': 40.64499,
 'pickup_longitude': -73.78115,
 'trip_distance': 18.38}
```

In [14]:
```
def location(trip):
    return list(v for k,v in trip.items() if k == 'pickup_latitude' or k ==
pass
```

In [15]:
```
first_location = location(first_trip) # [40.64499, -73.78115]
first_location # [40.64499, -73.78115]
```

Out[15]:
```
[40.64499, -73.78115]
```

Ok, now that we can turn a trip into a location, let's turn a location into a marker. Write a function called `to_marker` that takes in a location (in the form of a list) as an argument, and returns a folium `circleMarker` for that location. The radius of the marker should always equal 6.

In [16]:
```
def to_marker(location):
    circle_marker = folium.CircleMarker(location, radius=6)
    circle_marker.add_to(manhattan_map)
    return circle_marker
    pass
```

In [17]:
```
import json
times_square_marker = to_marker([40.7589, -73.9851])

times_square_marker and times_square_marker.location # [40.7589, -73.9851]
print(times_square_marker.location)
print(times_square_marker.options['radius'])
#times_square_marker and json.loads(times_square_marker.options)['radius']
```

```
[40.7589, -73.9851]
6
```

Ok, now that we know how to produce a single marker, let's write a function to produce lots. We can write a function called `markers_from_trips` that takes in a list of trips, and returns a marker object for each trip.

In [18]:
```
def markers_from_trips(trips):
    X = []
    for trip in trips:
        #X.append(location(trip))
        X.append(to_marker(location(trip)))
    return X
    pass

markers_from_trips(trips)
```

Out[18]:
```
[<folium.vector_layers.CircleMarker at 0x7f0ea1a8b208>,
 <folium.vector_layers.CircleMarker at 0x7f0ea1a8b0b8>,
 <folium.vector_layers.CircleMarker at 0x7f0ea1a8b630>,
 <folium.vector_layers.CircleMarker at 0x7f0ea1a8b5f8>,
 <folium.vector_layers.CircleMarker at 0x7f0ea1a8b588>,
 <folium.vector_layers.CircleMarker at 0x7f0ea1a8b7b8>,
 <folium.vector_layers.CircleMarker at 0x7f0ea1a921d0>,
 <folium.vector_layers.CircleMarker at 0x7f0ea1a8b828>,
 <folium.vector_layers.CircleMarker at 0x7f0ea1a8b2b0>,
 <folium.vector_layers.CircleMarker at 0x7f0ea1a8b710>,
 <folium.vector_layers.CircleMarker at 0x7f0ea1a8b668>,
 <folium.vector_layers.CircleMarker at 0x7f0ea1a8b550>,
 <folium.vector_layers.CircleMarker at 0x7f0ea1a8b978>,
 <folium.vector_layers.CircleMarker at 0x7f0ea1a8b5c0>,
 <folium.vector_layers.CircleMarker at 0x7f0ea1a8b748>,
 <folium.vector_layers.CircleMarker at 0x7f0ea1a8b470>,
 <folium.vector_layers.CircleMarker at 0x7f0ea1a8b6a0>,
 <folium.vector_layers.CircleMarker at 0x7f0ea1a8b6d8>,
 <folium.vector_layers.CircleMarker at 0x7f0ea1a8b780>,
```

In [19]:
```
trip_markers = markers_from_trips(cleaned_trips)
counter = 0
for trip_marker in trip_markers:
    counter += 1
    print(str(counter)+" location[latitude,longitude]:"+str(trip_marker.loc
```

```
7 location[latitude,longitude]:[40.800219, -73.968098]
8 location[latitude,longitude]:[40.648509, -73.783508]
9 location[latitude,longitude]:[40.721897, -73.983493]
10 location[latitude,longitude]:[40.791566, -73.972224]
11 location[latitude,longitude]:[40.744896, -73.978619]
12 location[latitude,longitude]:[40.721951, -73.844435]
13 location[latitude,longitude]:[40.732382, -74.001682]
14 location[latitude,longitude]:[40.768339, -73.961478]
15 location[latitude,longitude]:[40.775933, -73.962446]
16 location[latitude,longitude]:[40.794829, -73.971476]
17 location[latitude,longitude]:[40.758647, -73.964878]
18 location[latitude,longitude]:[40.713638, -74.011587]
19 location[latitude,longitude]:[40.77403, -73.874597]
20 location[latitude,longitude]:[40.728127, -73.99887]

21 location[latitude,longitude]:[40.759671, -73.976688]
22 location[latitude,longitude]:[40.772651, -73.967095]
23 location[latitude,longitude]:[40.770319, -73.960222]
24 location[latitude,longitude]:[40.720457, -73.988065]
```

In [20]:
```
cleaned_trips[0:4]
```

Out[20]:
```
[{'pickup_latitude': 40.64499,
  'pickup_longitude': -73.78115,
  'trip_distance': 18.38},
 {'pickup_latitude': 40.766931,
  'pickup_longitude': -73.982098,
  'trip_distance': 1.3},
 {'pickup_latitude': 40.77773,
  'pickup_longitude': -73.951902,
  'trip_distance': 4.5},
 {'pickup_latitude': 40.795678,
  'pickup_longitude': -73.971049,
  'trip_distance': 2.4}]
```

In [21]:
```
trip_markers and len(trip_markers) # 1000
print(len(trip_markers))
list(map(lambda marker: marker.location, trip_markers[0:4]))
# [[40.64499, -73.78115],
#  [40.766931, -73.982098],
#  [40.77773, -73.951902],
#  [40.795678, -73.971049]]
```

```
1000
```

Out[21]:
```
[[40.64499, -73.78115],
 [40.766931, -73.982098],
 [40.77773, -73.951902],
 [40.795678, -73.971049]]
```

Ok, now that we have a function that creates locations, and a function that creates markers, it is time to write a function to plot a map.

Write a function called `map_from` that, provided the first argument of a list location and second argument an integer representing the `zoom_start` , returns a `folium` map the corresponding location and `zoom_start` attributes.

> Hint: The following is how to write a map with folium:
>
> ```
>     folium.Map(location=location, zoom_start=zoom_amoun
> t)
> ```

In [22]:
```
def map_from(location, zoom_amount):
    location_map = folium.Map(location, zoom_start=zoom_amount)
    return location_map
    pass
```

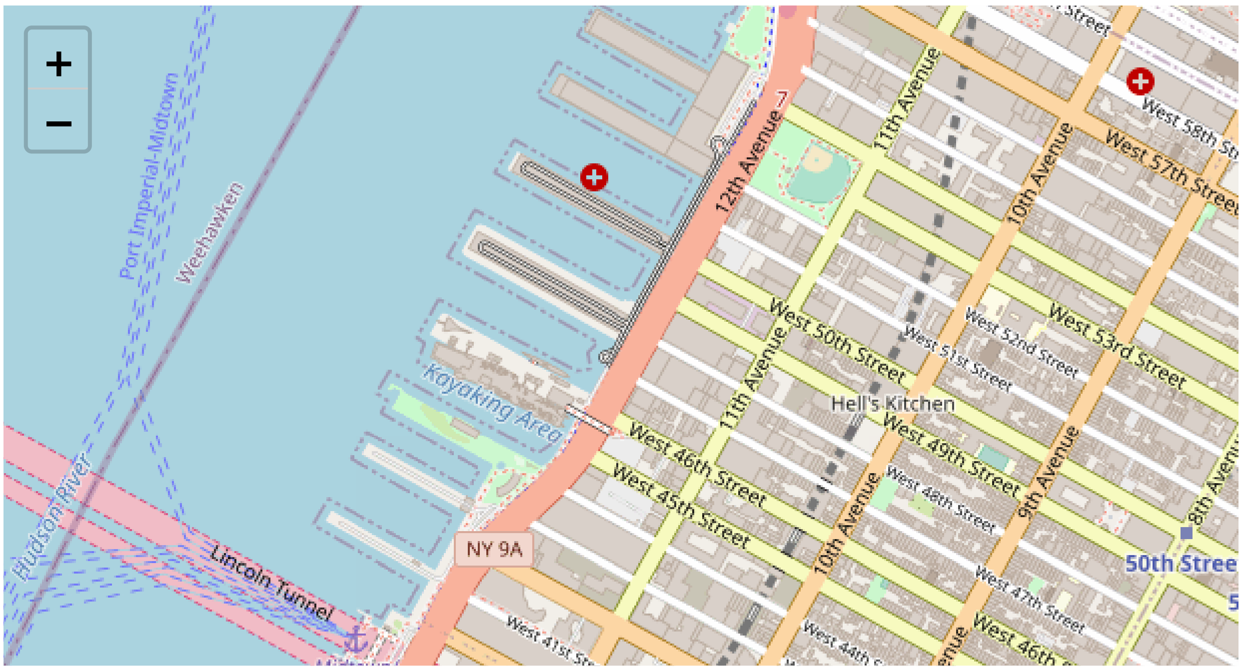
In [23]:
```
times_square_map = map_from([40.7589, -73.9851], 15)
print(times_square_map.location)
times_square_map and times_square_map.location # [40.7589, -73.9851]
#times_square_map and times_square_map.zoom_start # 15
```

```
[40.7589, -73.9851]
```

Out[23]:
```
[40.7589, -73.9851]
```

In [24]:
```
times_square_marker and times_square_marker.add_to(times_square_map)
times_square_map
```

Out[24]:



Now that we have a marker and a map, now let's write a function that adds a lot of markers to a map. This function should add each marker in the list to the map object then return the updated map object.

In [25]:
```
manhattan_map = map_from([40.7589, -73.9851], 13)
```

In [26]:
```
def add_markers(markers, map_obj):
    counter = 0
    for marker in markers:
        counter += 1
        #print("location:"+str(marker.location))
        location_map = map_from(marker.location, zoom_amount=15)
        if counter%25 == 0:
            print(str(counter)+" location:"+str(location_map.location))
        location_map.add_to(map_obj)
    return map_obj
    pass
```

In [27]:
```
map_with_markers = add_markers(trip_markers, manhattan_map)
```

```
25 location:[40.754789, -73.973606]
50 location:[40.741249, -74.005201]
75 location:[40.750722, -73.968447]
100 location:[40.74964, -73.972212]
125 location:[40.767446, -73.984105]
150 location:[40.753632, -73.988899]
175 location:[40.75331, -73.968992]
200 location:[40.759825, -73.970231]
225 location:[40.771385, -73.96467]
250 location:[40.702557, -73.93467]
275 location:[40.645322, -73.776652]
300 location:[40.746142, -73.984852]
325 location:[40.758985, -73.989435]
350 location:[40.659956, -73.99842]
375 location:[40.766804, -73.969071]
400 location:[0.0, 0.0]
425 location:[40.748193, -73.984715]
450 location:[40.76944, -73.952025]
475 location:[40.772255, -73.982507]
500 location:[40.727368, -74.0062]
525 location:[40.777252, -73.982625]
550 location:[40.769041, -73.988708]
575 location:[40.729574, -73.991874]
600 location:[40.70746, -74.004712]
625 location:[40.744275, -74.006706]
650 location:[40.742219, -73.994123]
675 location:[40.718208, -73.986443]
700 location:[40.74933, -73.97076]
725 location:[40.741115, -74.00577]
750 location:[40.78232, -73.951423]
775 location:[40.720975, -73.993653]
800 location:[40.737382, -73.997073]
825 location:[40.75781, -73.975233]
850 location:[40.740622, -74.007735]
875 location:[40.671487, -73.984354]
900 location:[40.770965, -73.964002]
925 location:[40.71032, -74.009677]
950 location:[40.757541, -73.974641]
975 location:[40.796757, -73.970537]
1000 location:[40.787172, -73.97763]
```

In [ ]:
```
map_with_markers
```

Using Nearest Neighbors

Ok, let's write a function that given a latitude and longitude will predict the distance for us. We'll do this by first finding the nearest trips given a latitude and longitude.

Here we once again apply the nearest neighbors formula. As a first step, write a function named `distance_location` that calculates the distance in pickup location between two trips.

In [28]:
```
import math

def distance_location(selected_trip, neighbor_trip):
    return math.sqrt(pow(selected_trip['pickup_latitude']-neighbor_trip['pi
    pass
```

In [29]:
```
first_trip = {'pickup_latitude': 40.64499, 'pickup_longitude': -73.78115, '
second_trip = {'pickup_latitude': 40.766931, 'pickup_longitude': -73.982098
distance_first_and_second = distance_location(first_trip, second_trip)

print(round(distance_first_and_second, 5))
distance_first_and_second and round(distance_first_and_second, 3) # 0.235
```

```
0.23505
```

Out[29]:
```
0.235
```

Ok, next write a function called `distance_between_neighbors` that adds a new key-value pair, called `distance_from_selected` , that calculates the distance of the `neighbor_trip` from the `selected_trip` .

In [30]:
```
def distance_between_neighbors(selected_trip, neighbor_trip):
    neighbor_trip_renew = dict()
    neighbor_trip_clone = neighbor_trip.copy()
    neighbor_trip_clone['distance_from_selected'] = distance_location(selec
    neighbor_trip_order = sorted(neighbor_trip_clone.keys(), key = lambda k
    for i in neighbor_trip_order:
        values = neighbor_trip_clone[i] # retrieve corresponding value to c
        #print(i+":"+str(values))
        #neighbor_trip_renew[i] = values        # method 1
        neighbor_trip_renew.update({i:values})  # method 2

    return neighbor_trip_renew
    pass
```

In [31]:
```
distance_between_neighbors(first_trip, second_trip)

# {'distance_from_selected': 0.23505256047318146,
#  'pickup_latitude': 40.766931,
#  'pickup_longitude': -73.982098,
#  'trip_distance': 1.3}
```

Out[31]:
```
{'distance_from_selected': 0.23505256047318146,
 'pickup_latitude': 40.766931,
 'pickup_longitude': -73.982098,
 'trip_distance': 1.3}
```

Ok, now our `neighbor_trip` has another attribute called `distance_from_selected` , that indicates the distance from the `neighbor_trip` 's pickup location from the `selected_trip` .

> ** Understand the data:** Our dictionary now has a few attributes, two of which say distance. Let's make sure we understand the difference.

- **`distance_from_selected`**: This is our calculation of the distance of the neighbor's pickup location from the selected trip.
- **`trip_distance`**: This is the attribute we were provided initially. It tells us the length of the neighbor's taxi trip from pickup to drop-off.

Next, write a function called `distance_all` that provided a list of neighbors, returns each of those neighbors with their respective `distance_from_selected` numbers.

In [32]:
```
def distance_all(selected_individual, neighbors):
    X=[]
    neighbors_clone = neighbors.copy()
    neighbors_clone = list(filter(lambda neighbor: selected_individual['pic
    neighbors_update = list(map(lambda neighbor:distance_between_neighbors(
    #for neighbor in neighbors_update:
    #X.append(neighbor['distance_from_selected'])
    for neighbor in neighbors_update:
        #X.append({v for k,v in neighbor.items() if k == 'distance_from_sel
        for k,v in neighbor.items():
            if k == 'distance_from_selected':
                X.append(float(v))
    return X
    pass

distance_all(first_trip, cleaned_trips[0:4])
```

Out[32]:
```
[0.23505256047318146, 0.2162779533470808, 0.24242215976473674]
```

In [33]:
```
cleaned_trips and distance_all(first_trip, cleaned_trips[0:4])
```

Out[33]:
```
[0.23505256047318146, 0.2162779533470808, 0.24242215976473674]
```

Now write the nearest neighbors formula to calculate the distance of the `selected_trip` from all of the `cleaned_trips` in our dataset. If no number is provided, it should return the top 3 neighbors.

In [34]:
```
def nearest_neighbors(selected_trip, trips, number = 3):
    trips_clone = trips.copy()
    trips_clone = list(filter(lambda trip:selected_trip['pickup_latitude']
    trips_update = list(map(lambda trip:distance_between_neighbors(selected
    trips_order = sorted(trips_update,key = lambda i:i['distance_from_selec
    if (number == number or number == len(trips_order)):
        trips_select = trips_order[:number]

    return trips_select
    pass
```

In [35]:
```python
new_trip = {'pickup_latitude': 40.64499,
'pickup_longitude': -73.78115,
'trip_distance': 18.38}

nearest_three_neighbors = nearest_neighbors(new_trip, cleaned_trips or [],
nearest_three_neighbors
# [{'distance_from_selected': 0.00045692887849187 92,
#   'pickup_latitude': 40.64483,
#   'pickup_longitude': -73.781578,
#   'trip_distance': 7.78},
#  {'distance_from_selected': 0.0011292165425673159,
#   'pickup_latitude': 40.644657,
#   'pickup_longitude': -73.782229,
#   'trip_distance': 12.7},
#  {'distance_from_selected': 0.0042359798158141185,
#   'pickup_latitude': 40.648509,
#   'pickup_longitude': -73.783508,
#   'trip_distance': 17.3}]
```

Out[35]:
```
[{'distance_from_selected': 0.00045692887849187 92,
  'pickup_latitude': 40.64483,
  'pickup_longitude': -73.781578,
  'trip_distance': 7.78},
 {'distance_from_selected': 0.0011292165425673159,
  'pickup_latitude': 40.644657,
  'pickup_longitude': -73.782229,
  'trip_distance': 12.7},
 {'distance_from_selected': 0.0042359798158141185,
  'pickup_latitude': 40.648509,
  'pickup_longitude': -73.783508,
  'trip_distance': 17.3}]
```

Ok great! Now that we can provide a new trip location, and find the distances of the three nearest trips, we can take calculate an estimate of the trip distance for that new trip location.

We do so simply by calculating the average of it's nearest neighbors.

In [36]:
```python
import statistics
def mean_distance(neighbors):
    nearest_distances = list(map(lambda neighbor: neighbor['trip_distance']
    return round(statistics.mean(nearest_distances), 3)

nearest_three_neighbors = nearest_neighbors(new_trip, cleaned_trips or [],
distance_estimate_of_selected_trip = mean_distance(nearest_three_neighbors)
distance_estimate_of_selected_trip
```

Out[36]: 12.593

Choosing the correct number of neighbors

Now, as we know from the last lesson, one tricky element is to determine how many neighbors to choose, our k value, before calculating the average. We want to choose our value of k such that it properly matches actual data, and so that it applies to new data. There are fancy formulas to properly matches actual data, and so that it applies to new data. There are fancy formulas to ensure that we **train** our algorithm so that our formula is optimized for all data, but here let's see different k values manually. This is the gist of choosing our k value:

- If we choose a k value too low, our formula will be too heavily influenced by a single neighbor, whereas if our k value is too high, we will be choosing so many neighbors that our nearest neighbors formula will not be adjust enough according to locations.

Ok, let's experiment with this.

First, let's choose a midtown location, to see what the trip distance would be. A Google search reveals the coordinates of 51st and 7th avenue to be the following.

In [37]:
```
midtown_trip = dict(pickup_latitude=40.761710, pickup_longitude=-73.982760)
```

In [38]:
```python
seven_closest = nearest_neighbors(midtown_trip, cleaned_trips, number = 7)
seven_closest
# [{'trip_distance': 0.58,
#   'pickup_latitude': 40.761372,
#   'pickup_longitude': -73.982602,
#   'distance_from_selected': 0.00037310588309379025},
#  {'trip_distance': 0.8,
#   'pickup_latitude': 40.762444,
#   'pickup_longitude': -73.98244,
#   'distance_from_selected': 0.00080072217404248},
#  {'trip_distance': 1.4,
#   'pickup_latitude': 40.762767,
#   'pickup_longitude': -73.982293,
#   'distance_from_selected': 0.0011555682584735844},
#  {'trip_distance': 8.3,
#   'pickup_latitude': 40.762868,
#   'pickup_longitude': -73.983233,
#   'distance_from_selected': 0.0012508768924205918},
#  {'trip_distance': 1.26,
#   'pickup_latitude': 40.760057,
#   'pickup_longitude': -73.983502,
#   'distance_from_selected': 0.0018118976240381972},
#  {'trip_distance': 0.0,
#   'pickup_latitude': 40.760644,
#   'pickup_longitude': -73.984531,
#   'distance_from_selected': 0.002067074502774709},
#  {'trip_distance': 1.72,
#   'pickup_latitude': 40.762107,
#   'pickup_longitude': -73.98479,
#   'distance_from_selected': 0.0020684557041472677}]
```

Out[38]:
```
[{'distance_from_selected': 0.00037310588309379025,
  'pickup_latitude': 40.761372,
  'pickup_longitude': -73.982602,
  'trip_distance': 0.58},
 {'distance_from_selected': 0.00080072217404248,
  'pickup_latitude': 40.762444,
  'pickup_longitude': -73.98244,
  'trip_distance': 0.8},
 {'distance_from_selected': 0.0011555682584735844,
  'pickup_latitude': 40.762767,
  'pickup_longitude': -73.982293,
  'trip_distance': 1.4},
 {'distance_from_selected': 0.0012508768924205918,
  'pickup_latitude': 40.762868,
  'pickup_longitude': -73.983233,
  'trip_distance': 8.3},
 {'distance_from_selected': 0.0018118976240381972,
  'pickup_latitude': 40.760057,
  'pickup_longitude': -73.983502,
  'trip_distance': 1.26},
 {'distance_from_selected': 0.002067074502774709,
  'pickup_latitude': 40.760644,
  'pickup_longitude': -73.984531,
  'trip_distance': 0.0},
 {'distance_from_selected': 0.0020684557041472677,
  'pickup_latitude': 40.762107,
```

```
  'pickup_longitude': -73.98479,
  'trip_distance': 1.72}]
```

Looking at the `distance_from_selected` it appears that our our trips are still fairly close to our selected trip. Notice that most of the data is within a distance of .002 away, so going to the top 7 nearest neighbors didn't seem to give us neighbors too far from each other, which is a good sign.

Still, it's hard to know what distance in latitude and longitude really look like, so let's map the data.

In [39]:
```
midtown_location = location(midtown_trip) # [40.76171, -73.98276]
midtown_map = map_from(midtown_location, 16)
closest_markers = markers_from_trips(seven_closest)

add_markers(closest_markers, midtown_map)
```

Out[39]:

Ok. These locations stay fairly close to our estimated location of 51st street and 7th Avenue. So they could be a good estimate of a trip distance.

In [40]:
```
mean_distance(seven_closest) # 2.009
```

Out[40]: 2.009

Ok, now let's try a different location

In [41]:
```
charging_bull_closest = nearest_neighbors({'pickup_latitude': 40.7049, 'pic
```

In [42]:
```
mean_distance(charging_bull_closest) # 3.145
```

Out[42]: 3.145

Ok, so there appears to be a significant difference between choosing a location near Times Square versus choosing a location at Wall Street.

Summary

In this lab, we used the nearest neighbors function to predict the length of a taxi ride. To do so, we selected a location, then found a number of taxi rides closest to that location, and finally took the average trip lengths of the nearest taxi rides to find an estimate of the new ride's trip length. You can see that even with just a little bit of math and programming we can begin to make meaningful predictions with data.