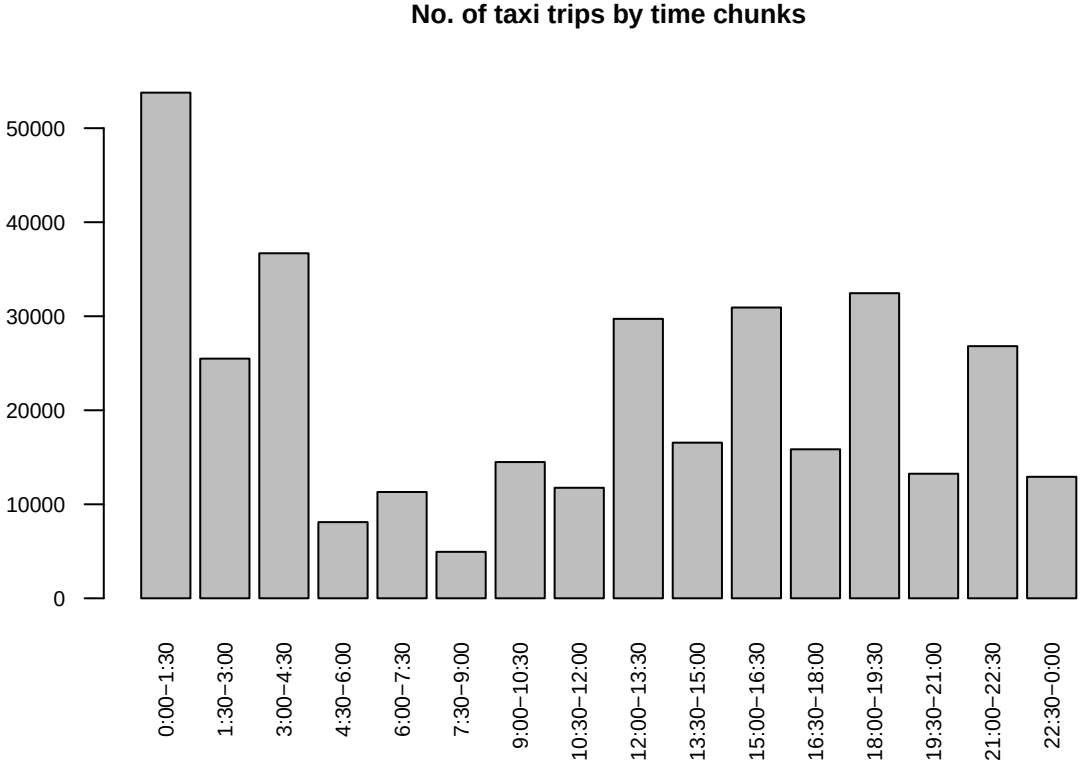
Taxis in Town

Evan Mata, Vu Phan, Nanut Chaichanawanich

May 29, 2018

1. Description of data set:

We began by downloading all the yellow cab taxi rides (New York city) in January 2016, the file size was around 1.7GB. Then we took 1 day worth of data from this dataset, and split it into 16 time chunks of 1.5 hours a chunk. After parsing the large data, and splitting it into chunks - the combined sized across all the chunks was 41.5MB. The smallest Chunk was 582 kB (5k rides) and the largest chunk: 6.3 MB (54k rides). The summary of the numbers of trips is given below:



We initially were going to chunk our data into 12 chunks, each one being the same time period everyday for a week. With 16 chunks, a week's data had a maximum chunk size 34.9 MB (290k rides) and a minimum chunk size of 3.3MB (27k rides). Considering that our 54k rides file took 2.33 hours to process, this means our 290k file would have been approximately $6\hat{}2 = 36$ times longer to process, or about 84 hours. On top of this, we initially were going to do 12 chunks of 2 hours each. So assuming our rides count scaled by exactly by this 16/12 factor, it would have taken approximately 150 hours ($1.33\hat{}2$). This also demonstrates how helpful simply changing chunk size was; a reduction from 16 to 12 saved us nearly a factor of 2 in time. However, there were some limitations on minimum chunk size for realistic data, as discussed later.

2. Hypotheses:

We hypothesize that our algorithm that identifies intersections of trips (described in greater detail in the next section) will show insight into the spatial as well as temporal distributions of traffic in New York city on new year day. Spatially, we expect places of celebration and festivity, such as the downtown area, to have a

higher volume of traffic (more intersection points) than less central areas. Temporally, we expect traffic to be high right after midnight, go down in the morning hours (people sleeping in after staying up late for new year celebrations) and pick up again later in the day.

3. Algorithms:

Our algorithm looks at all pairwise combinations of taxi trips in a given period of time and calculates their intersections. If a chunk has n lines, then the complexity of the algorithm is n^2. If we have m chunks, then the total complexity of the algorithm for the project is mn^2. Note that if we did a week of data, one chunk would have the information for every day of the week during its time bounds.

4. Big Data approaches:

We used mapreduce for our project. The main bulk of the work is one in the mapper, where each line which is read into the mapper is compared to the rest of the trips in the csv to find the intersection points. We represented each trip by a line segment and we find the intersection of those line segments; if two lines overlap, we return only a single value instead of an infinite numbers of intersection. Furthermore, we also checked whether the in-time and out-time of the trips overlap or not, we did not count the intersection if the times did not overlap. We then rounded the intersection points to 3 decimal places (we have calculated that this is fine enough of a mesh to be able to generate a good heatmap). We then used the combiners and reducers to sum up the number of rounded intersection points, these count is used as weights for each of the points in the heatmap later on.

5. Why a single machine would not work:

As noted in our discussion on algorithms, the number of computations within a chunk scaled as n^2, with n being the number of lines within that chunk. One could naively assume that they could reduce the chunk size to a point where n^2 is small, and then perform calculations on large amounts of chunks. However, this ignores real world issues, namely that a chunk is currently 1.5 hours of time, so if one reduces chunk sizes too far then they will have a time chunk that is smaller than a taxicab trip length. At small chunk sizes, taxicab rides would not overlap, and we would not get meaningful dataful. This establishes a lower bound on chunk sizes. As we found that the average trip length was 16 minutes, a 1.5 hour chunk size should be sufficient to cover a majority of trips, though there is some room for reducing the chunk size, we believe 1.5 hour is close to the optimal chunk size.

Having established a minimum chunk size as a necessity, we can extrapolate the time it would take a single core machine to run a chunk of our data. The google data center we used allowed us to use a maximum of 24 cores, so we had one master core and then 5 sets of 4 core machines. This enabled us to run our code ~20 times faster than on a single core machine.

Considering that our single largest chunk, for a single day's data, took ~2.33 hours to run with 20 cores, this would take ~46 hours on a single core machine. Doing the entire day's chunks would therefore take on the order of a week of constant computation. Instead, we were able to have everything processed within a day. These time savings are compounded in other ways too, such as debugging. For instance, some bugs only appeared once a file had mostly been processed. Instead of being able to debug these within an hour or two, it would take two days per issue. From these enormous time savings, we can clearly see the advantage multiprocessing offers.

6. Challenges:

How to chunk

There were many ways we could chunk up the data. We thought about using a map-reduce approach as it might increase the speed of the chunking if we have many nodes working on it at the same. Also, we thought we could run this in step one and yield the results to the next step right away, so we would not have to import the file again. However, we realized we could not find a way to split the chunks into a separate file, so if this method was used, a huge file would be generated with the index for the chunk as a column. When we find the intersection, we have to loop over each row and check whether the two trips are from the same chunk; which is extremely costly in large dataset. Therefore, we realized that the simplest way was to write a simple script instead.

Getting dataproc to work on Google compute engine VM

Since the beginning we developed our code on our local VM. We used the csv library to read and write our files extensively. Our first problem was that we used the csv library to write our output from a reducer to our file, but this only works when we have 1 reducere which doesn't leverage mapreduce. We then moved on to mr3px library, which allows us to export yield statements to csv directly (but somehow this didn't work when we transferred data to dataproc, and so in the end we just exported as .txt and parsed it later with a command written in cat).

Uploading files to dataproc (Our biggest problem)

In the beginning, when each line is fed into the mapper we would use the csv library to access the file and make the comparisons. However, after moving to dataproc we realized that there was no way to access a local file. Fortunately, the library included –file which allowed us to upload files we need on to dataproc. However, after implementing this, we keep getting an unknown error (no information provided by dataproc).

After reading the documentation, we realized that we cannot access the files directly in the mapper but have to initialize it in mapper_init before we can use it. We then found that, we cannot just initialize a csvreader object in the mapper_init. It was then, we realized that if we only have <100K rows in a chunk, we could store the whole csv in memory in the form of a list (this would solve the csv problem and would also optimize speed). We eventually did this, the code ran and became much faster.

How to run map reduce faster?

So after getting the dataproc running, we noticed that the mapreduce still took significant amount of time to run. Therefore, to speed this up we tried to run on bigger clusters using the command –instance-type X –num-core-instances X, to increase the number of instances and cores in each instance. We tried to opt for lots of core and lots of instance, but Google locked the total numbers of core we can use to 24, so we had to work within this quota (which is suboptimal, but requesting more quotas takes too much time). Therefore, we used the command –instance-type n1-highcpu-4 –num-core-instances 5 to run our dataproc.

We think that this is the best combination since this is 20 cores in the worker nodes, and 1 in the master node (if we did 3 instances of n1-highcpu-8, which is more efficient - we would have 24 cores in worker and 1 in master which google won't allow). We also opted for the faster cpu in hopes that each instances can have more nodes.

Heat map

To represent our data, we chose to do it in a heatmap. In the beginning, our mapreduce yielded all intersection points (not rounded to 3 decimal places), and so using geojson we see which neighbourhood each point fits in to generate a choropleth as we did last quarter; this requires looping over all the 1500+ shapefiles denoting each census district in New York. For ~1000 intersection points it took over 10 minutes to generate this using the Folium library, which is not scalable to larger dataset. We then tried to plot all of the intersection points onto the heatmap as this bypasses having to check for neighborhood we thought this would be faster, but because we are working with millions of intersection points, the library could not handle this.

In the end, the best approach we chose to go with is to round the intersection points to 3 decimal places and group them up by that in during the mapreduce itself, and using the counts as weights for the heatmap. Although the library documentation says that Folium supports weighted heatmaps, the documentation was outdated and it always gives an error. After hours of trying, we decided to switch to gmaps - the only other library available for python weighted heatmaps. We tried running this on the VM and hoped that it would output .html heatmap, but the exact code for .html export specified in the documentation also did not work. The only other way this library supports is through jupyter notebook, which is what we opted for in the end.

Original project

The scope of our original project turned into a challenge when we realized how quick it could be. Simply put, big data was not necessary. The project was to analyze an entire year or two's rides of taxi data, and find patterns in the taxiride distribution for a given location. This only essentially involved counting rides for a given location, though the input files would sum to 10-30Gb, depending on how long we aggregated our data over. However, this was 1-2 years worth of data broken into month sized datasets. Processing a single month on a CSIL computer took approximately a minute. This clearly did not need to leverage the power of multiprocessing. The challenge was figuring out how to take the code and ideas from this project and turn them into something more complex.

7. Results:

We have several noticeable results, which get more interesting when compared to a population density map. First, we can establish that our maps make sense. They have relatively few intersections in central park, as well as dense areas near the airport (we can see in the zoomed out version of the map the location of LaGuardia and JFK clearly). We also noticed that the airports become extremely busy during the 8th chunk (noon) and stays very busy till around the 14th chunk (10:30pm).

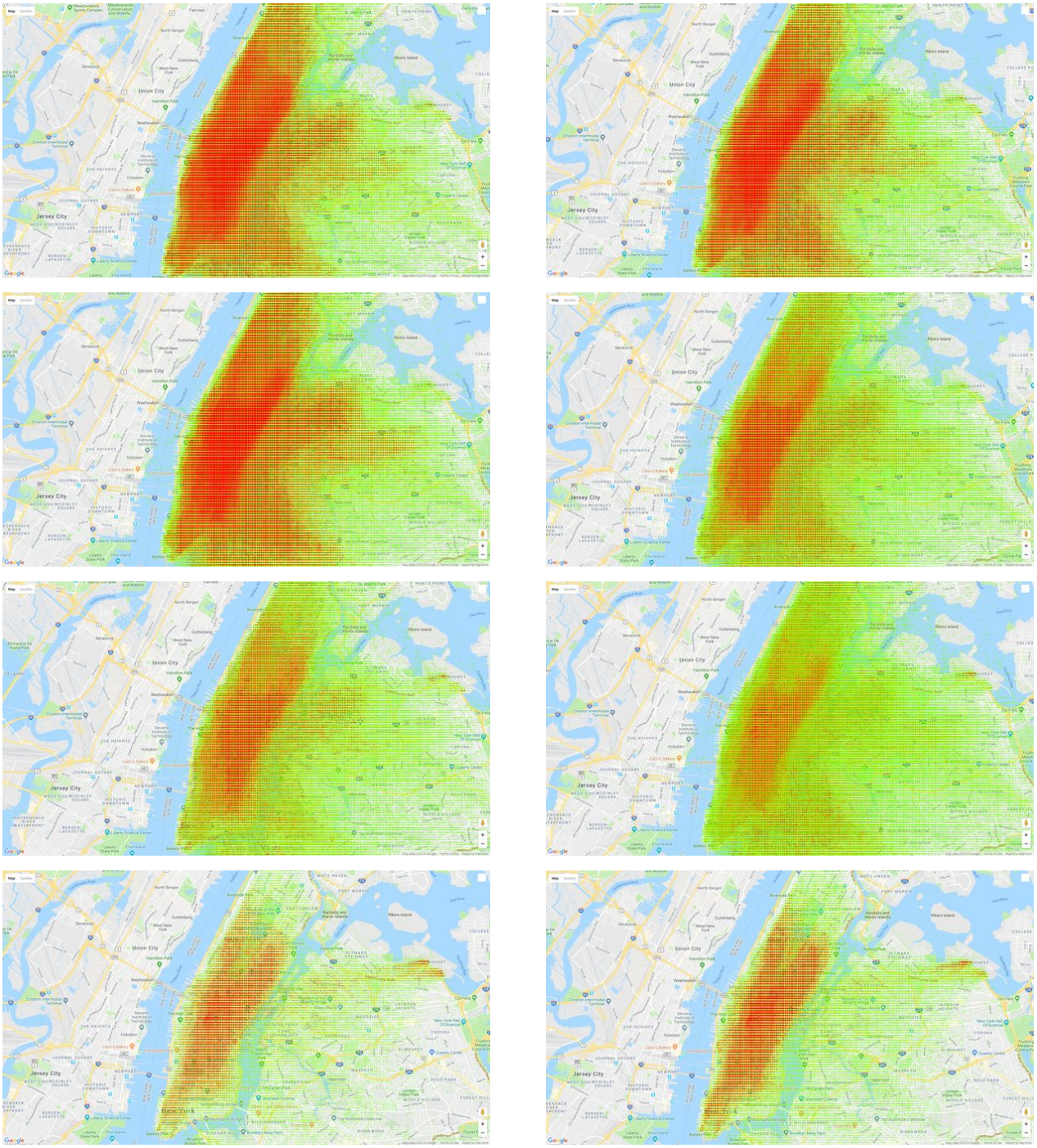
In addition, consider that we used the data from New Year day. The expectation is that there would be many taxi rides immediately after midnight compared to say 6-8am. Looking at the GIFs produced, we see that there is a large number of cars from 0:00-3:00am which supports our expectation. So the intensity of the traffic slowly dies down from midnight to around 2pm (we think that people went out during the night and want to sleep in), and then the intensity of traffic increase again.

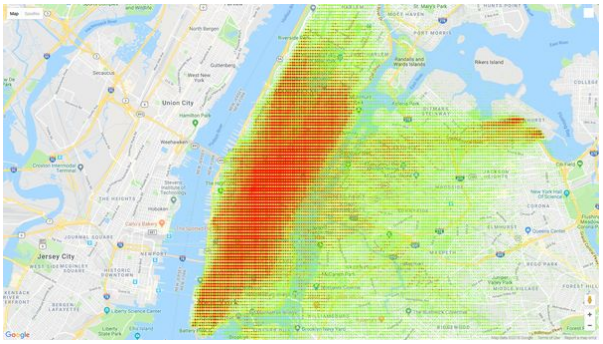
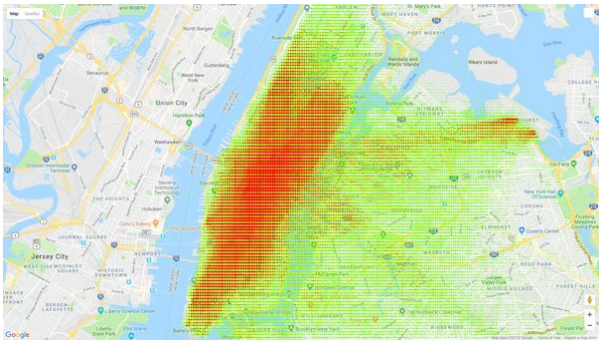
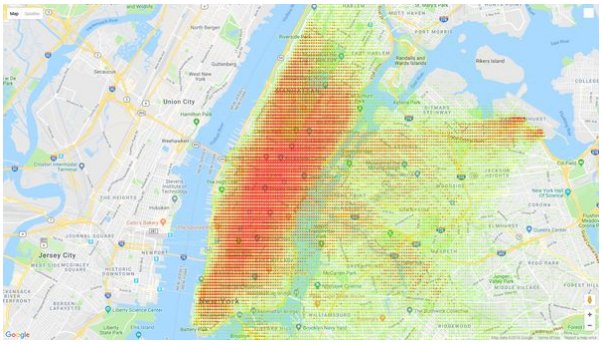
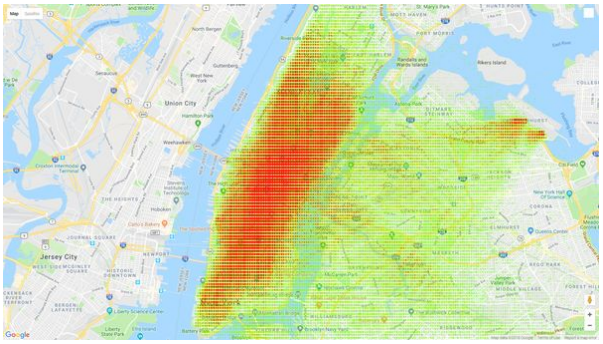
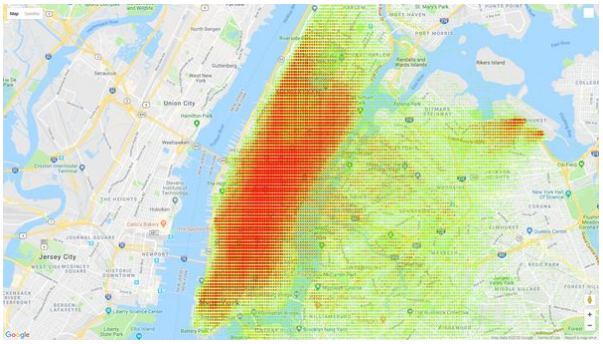
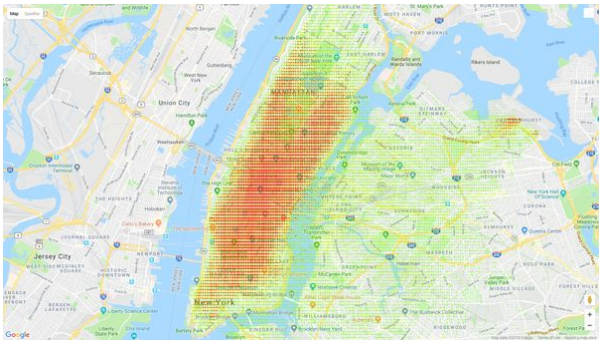
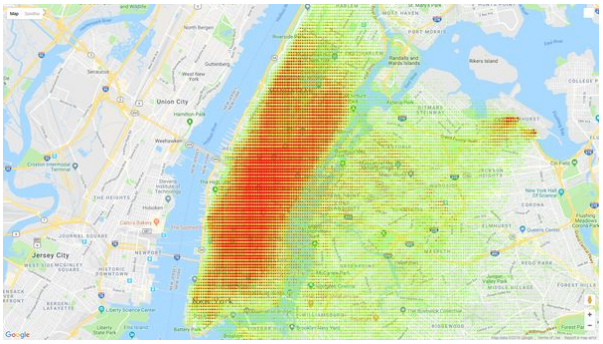
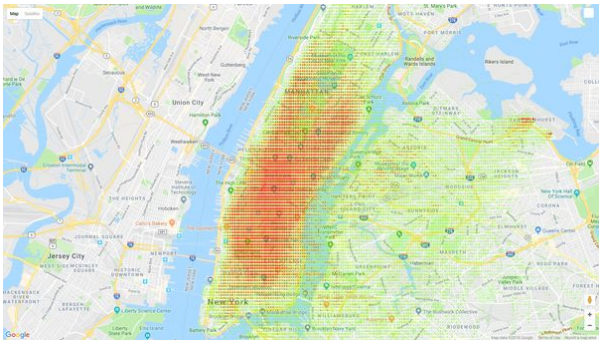
However, there are other interesting features. Despite the North West of Manhattan having a population density relatively close to that of Eastern Manhattan, there are significantly more taxi rides in the Eastern Manhattan area. This likely indicates that North-western Manhattan has better road networks for its population relative to the larger population of the Bronx which requires more infrastructure to support it. There is a clear decrease in area of dense traffic, right after midnight dense traffic is prevalent in all boroughs of New York, but as we move through the chunks we can see that it become confined to the Manhattan area.

All in all, our results are sensible and provide new insights into the temporal and spatial distribution of traffic after new year's day, a major celebration.

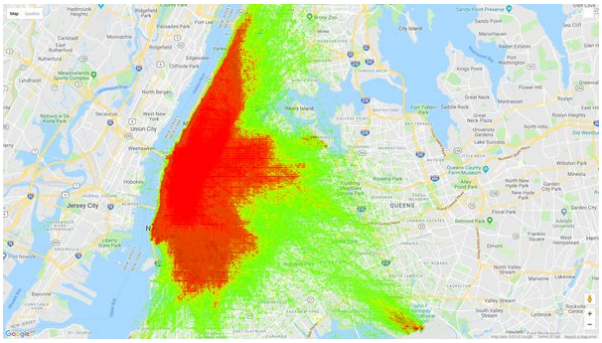
Heat maps are arranged chronologically left to right, top to bottom.

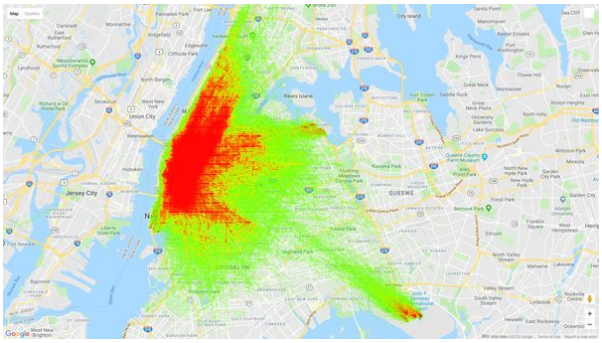
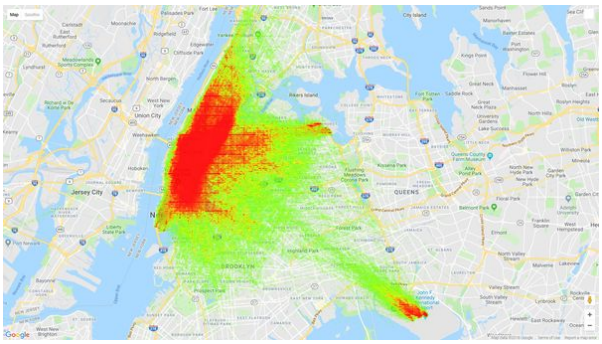
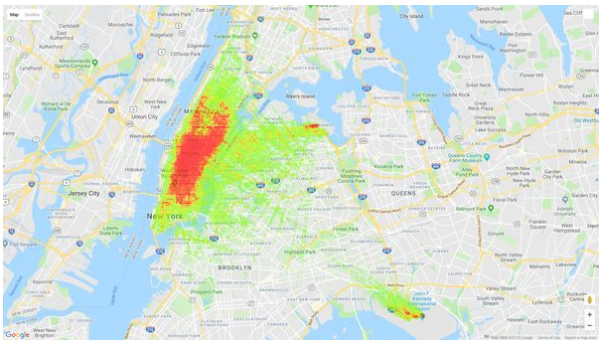
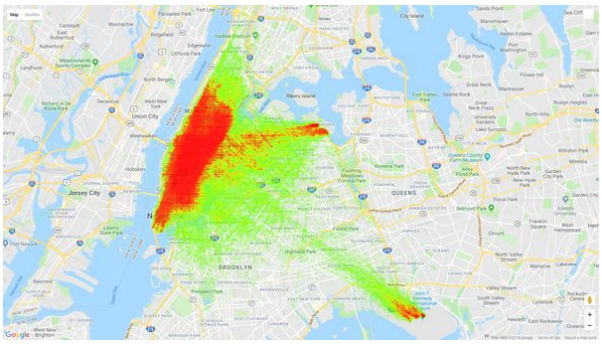
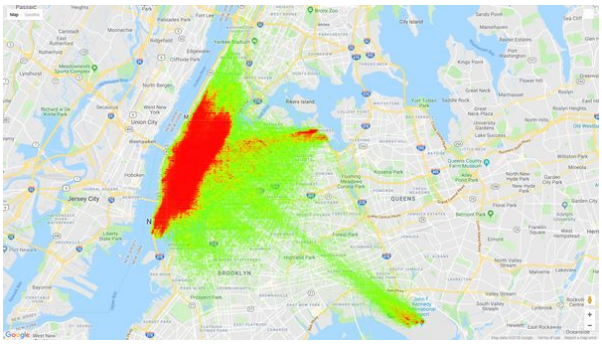
Zoomed in heat maps

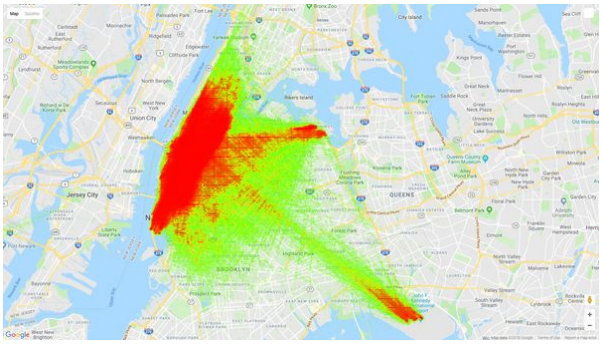
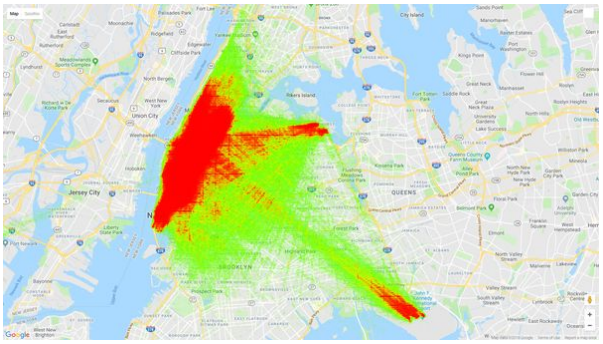
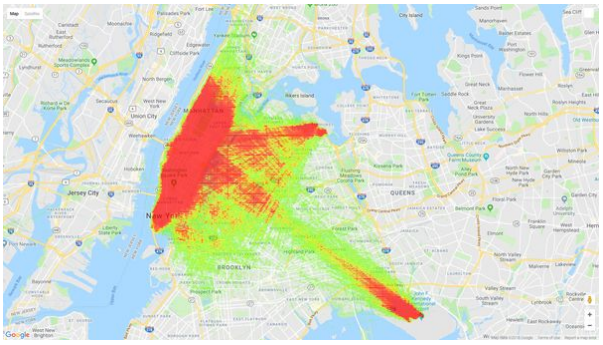
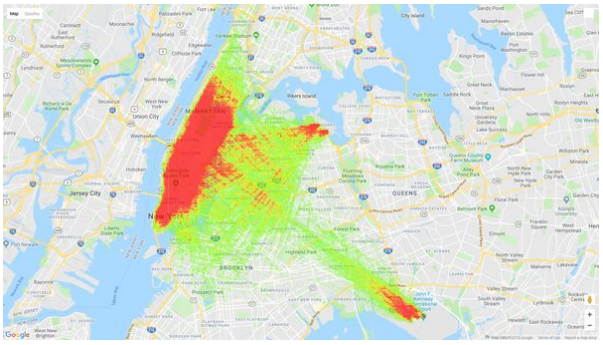
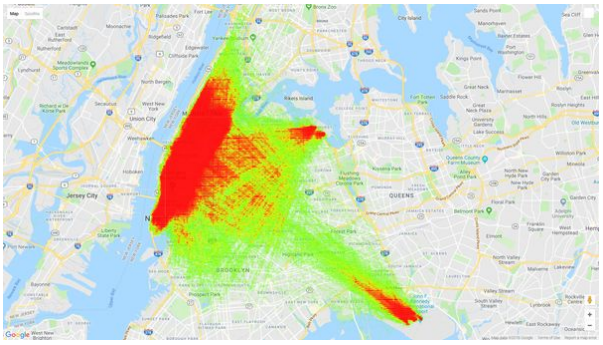
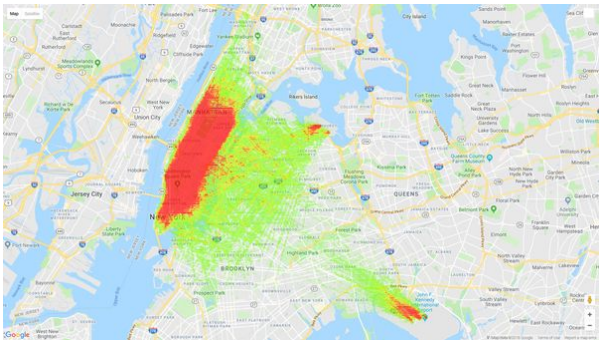
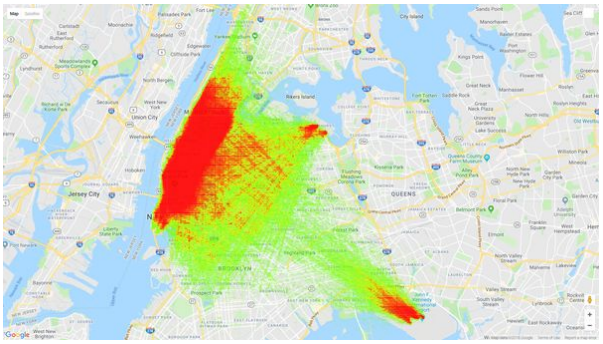
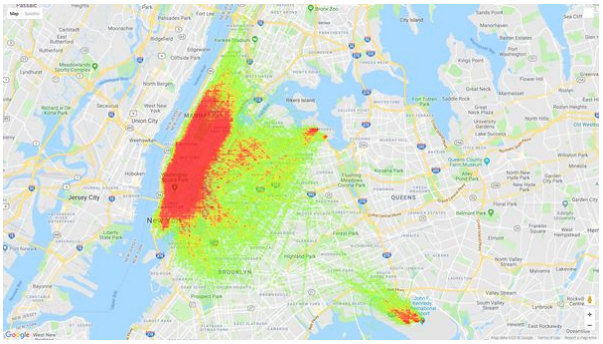


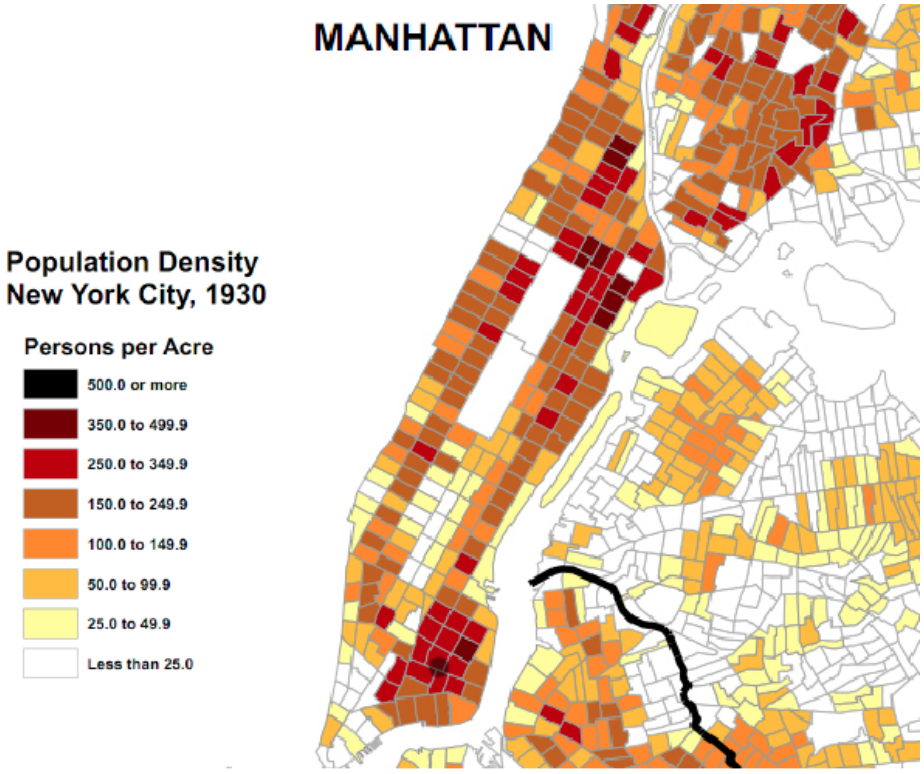
Zoomed out heat maps



Population density map



Source: https://bnhspine.com/new-york-city-population-density-map.html