480 Final Report on Ridesharing Results

Eric Ablang, Michael Mulligan, Anthony Venturella, Jia Kuang

Table Of Contents

Introduction

Taxi rides are expensive and the mileage driven contributes to climate change. Ride-sharing can mitigate both negative impacts. In a ride sharing mechanism, the passenger submits a ride request shortly before departure specifying some parameters such as pickup time, trip origin and destination, willingness to ride-share, the maximum delay tolerated due to ride-sharing. In our project, we were tasked with querying the trips database in SQL and determining how trips can be combined in an optimal way, i.e. in a way that maximizes the number of miles saved. At most 2 original trips can be combined into a single ridesharing trip. The taxi maximum occupancy is assumed to be 3.

We ended up processing rides for the entire year of 2015, but only in the region of Manhattan. This is due to the fact that there was a mix up when we originally read the project description, and by the point we realized we needed to process all of NYC we had already run about 7 months of the data just for Manhattan. We figured at this point if we started to include all of NYC in our algorithm for the last 5 months, the data would be skewed from the months we had already run. To try and resolve this, we ran the month of January on all of NYC in addition to having the entirety of 2015 run for just Manhattan. All of the figures and information you see throughout the report will be done for Manhattan, unless noted otherwise. There is a section dedicated to a comparison of January (Manhattan) against January (all of NYC).

The database querying strategy

Query and data cleaning

For our original query we only wanted to retrieve certain information from both of our indexed yellow data and green data mySQL databases. To do this, the query we sent to our database was to only retrieve the pickup and dropoff times and locations, and the amount of passengers in each ride. We also specified to only retrieve information in a certain time window on a desired date, we chose a 5 minute time window for our project.

```
SELECT Lpep_pickup_datetime, lpep_dropoff_datetime ,Pickup_longitude, Pickup_latitude,
Dropoff_longitude,
Dropoff_latitude, Passenger_count FROM green_data
WHERE Lpep_pickup_datetime >= '2015-01-01  00:00:00' AND Lpep_pickup_datetime <= '2015-01-01 00:05:00'
UNION
SELECT tpep_pickup_datetime, tpep_dropoff_datetime, pickup_longitude, pickup_latitude,
dropoff_longitude,
dropoff_latitude, trip_distance, passenger_count FROM yellow_data
WHERE tpep_pickup_datetime >= '2015-01-01  00:00:00' AND tpep_pickup_datetime <= '2015-01-01 00:05:00'
ORDER BY Lpep_pickup_datetime;
```

Figure 0: *Query for a 5min window*

After we got the rows from this query, we stored it into a pandas dataframe in our jupyter notebook and did some data cleaning. We first removed any rows that had missing information and also any trips that started and ended at the same position. After this, we created two new structures that stored the trip's distance and how long it took. For the distance, we calculated the haversine distance (in miles) between the trip's pickup and dropoff location. For the total time data, the difference in time between the pickup and dropoff time was calculated and stored in seconds.

Restricting rides to Manhattan

Due to a mixup in the project handout, we ran our algorithm on trips that originated and finished only in Manhattan. To do this, we first had to compare each trip's pickup and dropoff coordinates to a geojson file of Manhattan. We retrieved the border points of the city, created a polygon of those latitude and longitude coordinates, and then only kept trips that appeared within this polygon. The time window of these trips in the heat maps below are on 1/1/2015 from 00:00:00 until 00:05:000.

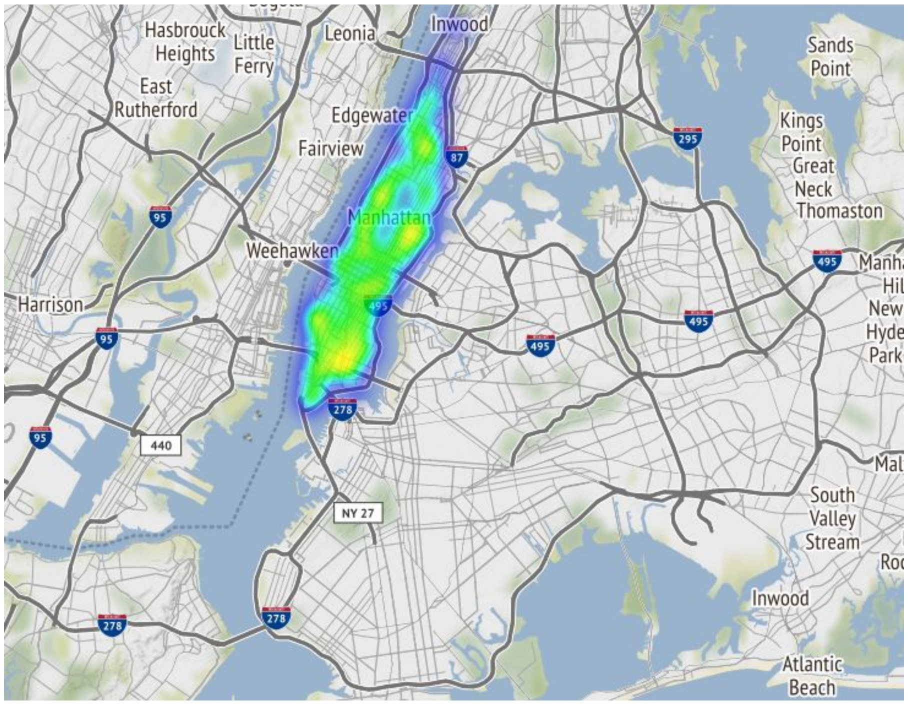


Figure 1: *Trips originating and ending only in Manhattan.*

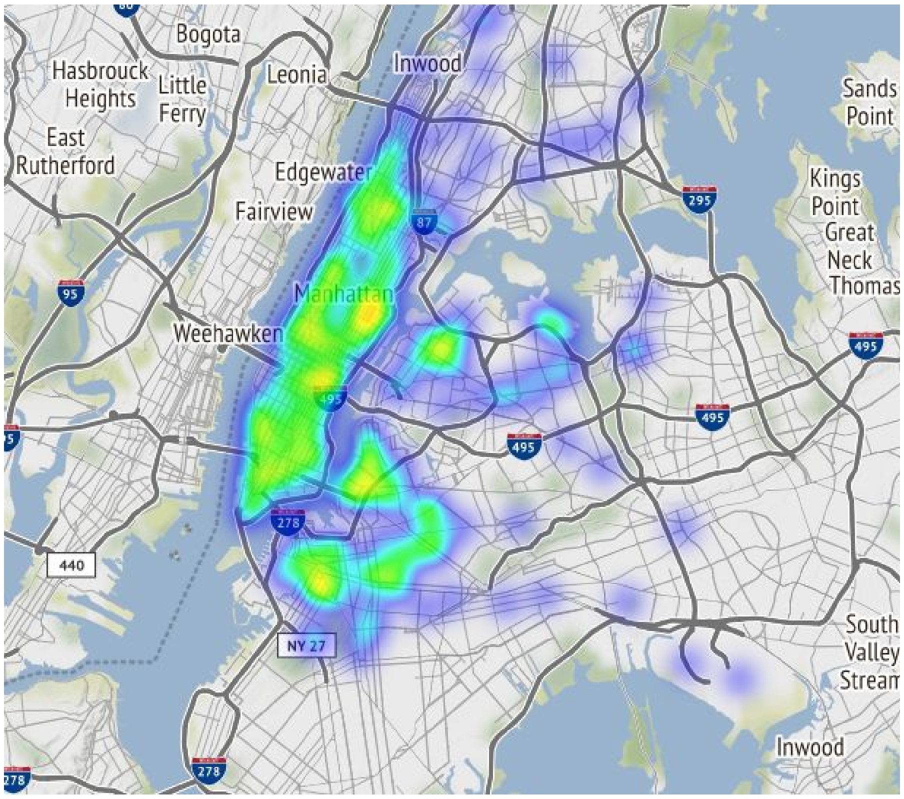


Figure 2: *Trips that occurred in all of NYC.*

From the information above, you can see that the majority of the rides are occurring in Manhattan in a close proximity, which gives a greater opportunity to have taxis participate in ridesharing.

Pooling Algorithm

The pooling algorithm we ran on our query window was sourced from the project handout. Here is a generalized pseudocode look at what the finished product ended up looking like. The full implementation can be viewed in the linked jupyter notebook at the end of the report. The sequence function included in the pseudo code returned the mileage traveled in the inputted sequence, and also if the specified delay was satisfied between the trips.

```
for i in range(0, rides):
    if (rides[i] >= 3 passengers):
        continue
    distance_one = haversine(rides[i])
    speed_one = distance_one/ (rides[i] trip time/60)

    for j in range(i, rides):
        if (rides[i] + rides[j] > 3 passengers OR i == j):
            continue
        distance_two = haversine(rides[i])
        speed_two = distance_two/ (rides[j] trip time/60)
        taxi_speed = speed_one + speed_two / 2
        seq_one_dist =    seq(o1___o2____d1______d2)
        seq_two_dist =    seq(o1___o2____d2______d1)
        seq_three_dist =  seq(o2___o1____d1______d2)
        seq_four_dist =   seq(o2___o1____d2______d1)
        find minimum sequence
            Create graph edge between min trip and set weight as distance
```
Figure 3: *pseudocode for the pooling algorithm*

As you can see from the pseudocode listed above, the most efficient path between the two points would be stored and have a graph edge created between those trip nodes with the weight being the distance. This graph would then have the maximum matching algorithm run on it.

Maximum Matching Algorithm

The maximum matching algorithm we decided to use was the built in variation in the NetworkX python graph library which can be found here.

The average query run time

Discuss how the average was calculated.

The average query runtime to retrieve results from a 5 minute window on our yellow and green databases was 250ms. This quick retrieval time was due to the fact that we had created an index on both our yellow and green taxi data. After we got the ride data via a union between our yellow and green databases, we then had to clean it and run the pooling algorithm. The cleaning of the data all occurred in O(n) time and was nearly instantaneous, but our major bottleneck was the pooling. Since this algorithm had to compare every ride with each other, the runtime was O(n^2) where n was the number of trips. After we had created our possible pool of trips and added them to a graph, we had to run the maximum matching algorithm which occurred in O(n^3) time, where n was the number of nodes. To process multiple months of data, we were able to split up the algorithm to run it in blocks of one week. Here is the time it took to run various amounts of taxi data, based on an average of each week that we ran.

	One Week	One Day	One Window
Time to Run	26 hours 30 minutes	3 hours 45 minutes	47 seconds

Figure 4: *Average time for algorithm to process different sized chunks of data after query*

The effectiveness in terms of mileage saved

Discuss how it was calculated.

The mileage saved was calculated based on the difference in total mileage traveled before we ran any algorithms, and total mileage traveled after rides were linked together via the ride-sharing algorithm. The algorithm we used is listed in the code in full, but a generalized look at the total ride calculations is as follows:

- First, create a set of ALL trip indices that were in the 5 minute window
- Loop through list of tuples that contain trip indices that will be ride shared
    - Remove the two merged trips indices from the set of all indices
    - Find distance between these two merged indices from previous map
    - Add that distance found between them to running total
- At this point, the only trips left in the set are trips that were NOT merged
- Loop through set, and add the miles from the lonesome trips to running total

After running this calculation algorithm on all of the data for 2015, we compiled it into a graph to visualize the savings. Figure 5 below contains the data for total trips traveled before our ride sharing algorithm, and after it’s implementation.

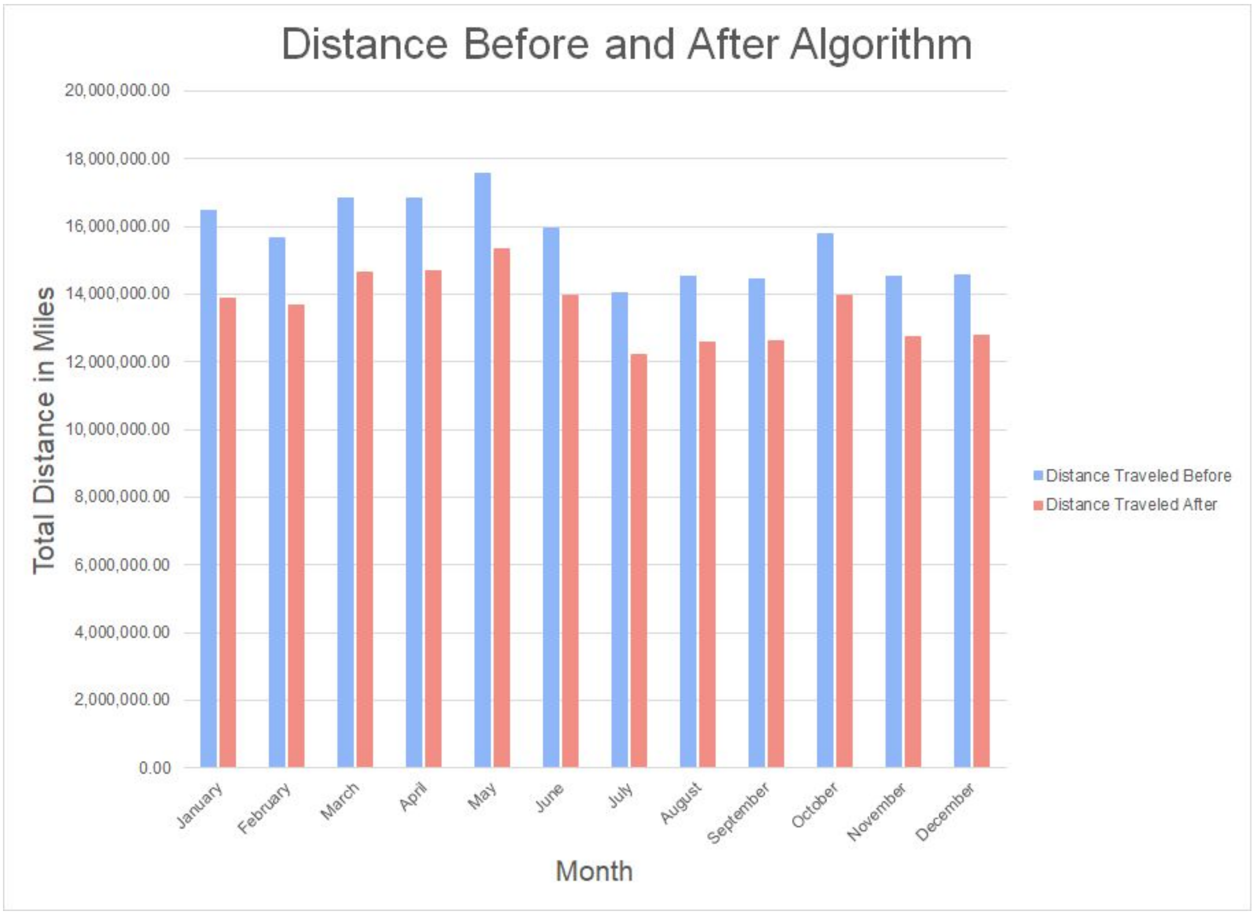


Figure 5: *Graph of Distance before and after Algorithm*

Figure 6 below gives a look at the total percentage of miles that were saved due to the merging of the rides, displaying the data for each individual month. The chart listed below in Figure 7 gives a more detailed look into the numbers we calculated for each month, and gives a total for the entire year of 2015.

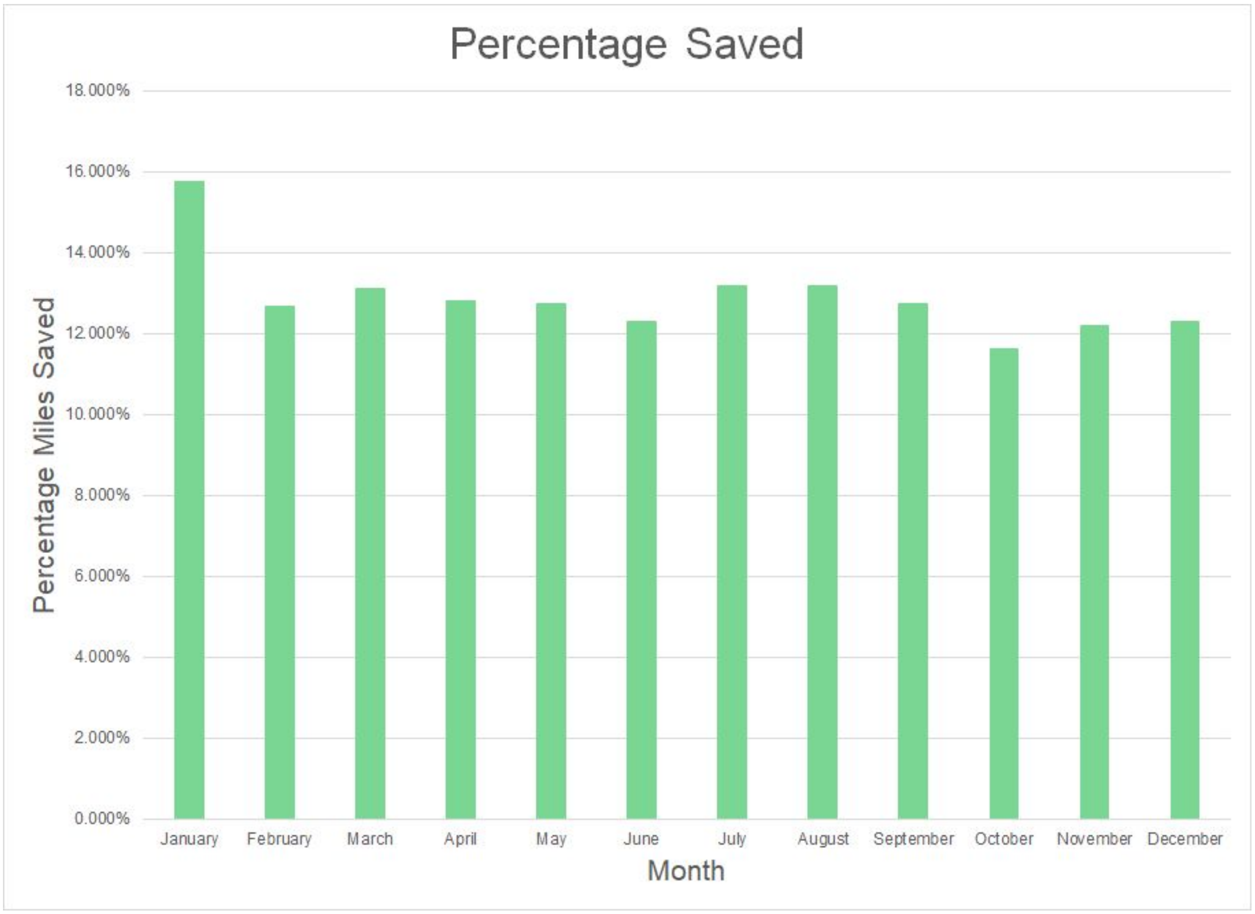


Figure 6: *Graph of Distance before and after Algorithm*

	Distance Before	Distance After	Total Saved	Percentage Saved
January	16,478,418.22	13,882,550.52	2,595,867.69	15.753%
February	15,640,098.23	13,657,657.08	1,982,441.15	12.675%
March	16,854,337.21	14,645,152.71	2,209,184.50	13.108%
April	16,826,365.24	14,669,840.11	2,156,525.13	12.816%
May	17,583,013.10	15,344,438.62	2,238,574.47	12.731%
June	15,931,719.01	13,972,121.08	1,959,597.93	12.300%
July	14,055,509.44	12,202,154.99	1,853,354.44	13.186%
August	14,513,725.19	12,597,934.59	1,915,790.61	13.200%
September	14,452,839.91	12,610,651.04	1,842,188.87	12.746%
October	15,799,634.48	13,963,176.61	1,836,457.87	11.623%
November	14,515,770.91	12,744,898.87	1,770,872.04	12.200%
December	14,558,486.87	12,766,789.39	1,791,697.48	12.307%
Totals	187,209,917.81	163,057,365.62	24,152,552.19	12.901%

Figure 7: *Table of distances traveled before and after ridesharing and percentage saved for the year by month*

For the results, we saved a total of 12.9 percent of miles for the data set where we only included rides from Manhattan. This is a significant amount of mileage saved since 12.9 percent of mileage can equate to millions of dollars saved on gas and thousands of hours saved. Overall, we felt that our algorithm was pretty efficient, since we were able to save a good amount of mileage, based on our results.

	Distance Before	Distance After	Total Saved	Percentage Saved
Manhattan January	16,478,418.22	13,882,550.52	2,595,867.69	15.753%
NYC January	28,425,412.41	25,848,375.15	2,577,037.26	9.066%

Figure 8: *Table comparing results of the different January runs*

For the results on the month of January where the month only had Manhattan data, the total mileage saved 15.8%. For the results on the month of January, where the month had the entire NYC data, the total mileage saved was 9.1%. This decrease in percentage made sense because after accounting for the other rides in the other boroughs, the pickup window, delay, and the locations of the rides may all have contributed to the efficiency of mileage saved and made tradeoffs. For example, the bulk of the taxi rides occur in Manhattan. Due to the smaller size of the borough, naturally more rides will be able to be merged together because of the density. Overall, the total percentage of mileage decreased but the area of data we accounted for dramatically increased.

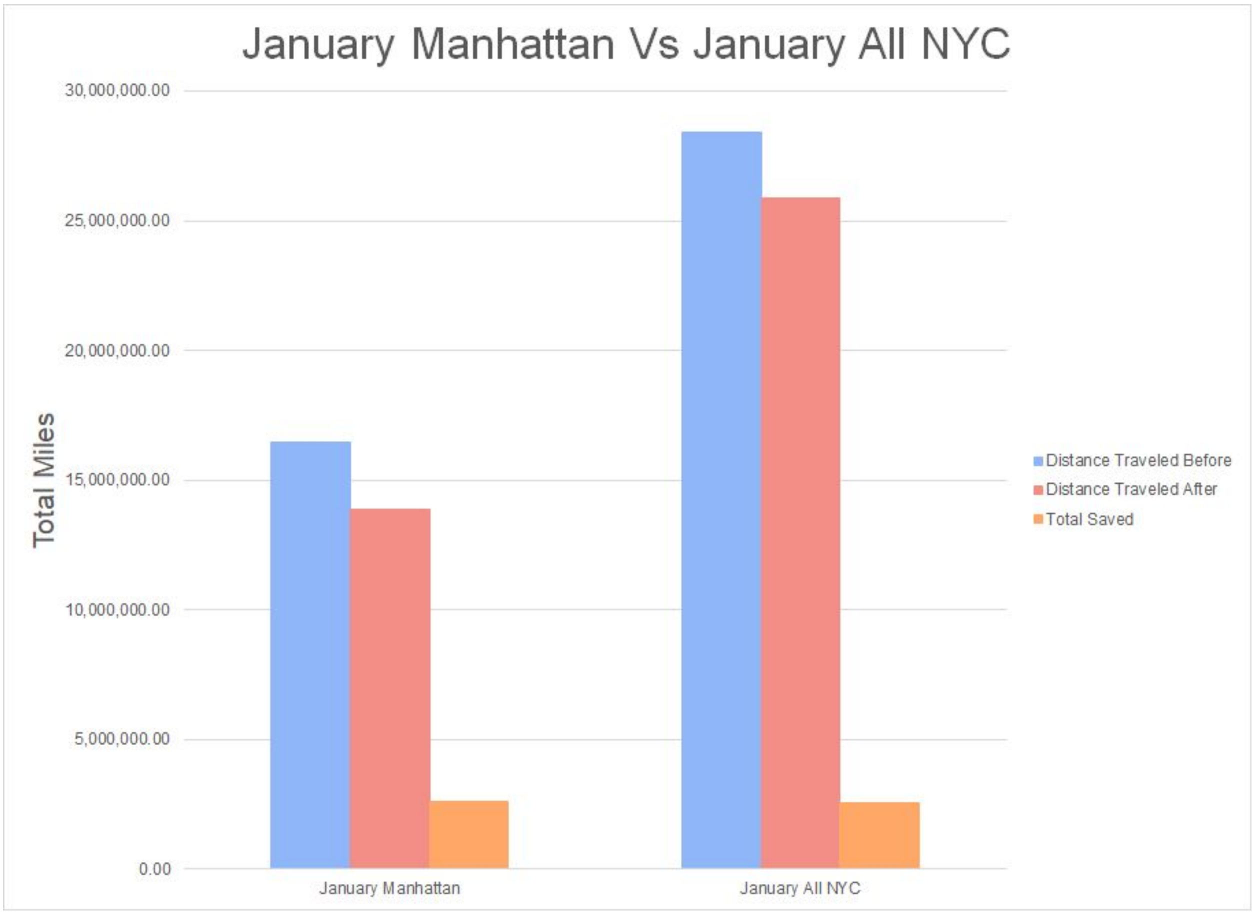


Figure 9: *Graph of total miles saved in just Manhattan vs all of NYC*

The period of time for which ride-sharing was analyzed

Indicate whether full or partial months were considered

The time window we chose to analyze was a 5 minute window. On average, the ride sharing algorithm produced a result in about 2 minutes 7 seconds in the 5 minute time window. We also ran our algorithm in a 2 minute and a 10 minute window. On average, our ride sharing algorithm took about 18 seconds with about 8.4% miles saved in the 2 minute time window and 10 minutes 52 seconds with about 12.5% of miles saved in the 10 minute time window.

We analyzed the full year of 2015 and used all green/yellow taxi data in our ride sharing algorithm. Throughout the year, on average the ride sharing algorithm was able to save about 12.9% of miles with having a total of 24,152,552 miles saved. We also chose to analyze all the trips originating and ending in Manhattan and all the trips that happened in New York City.

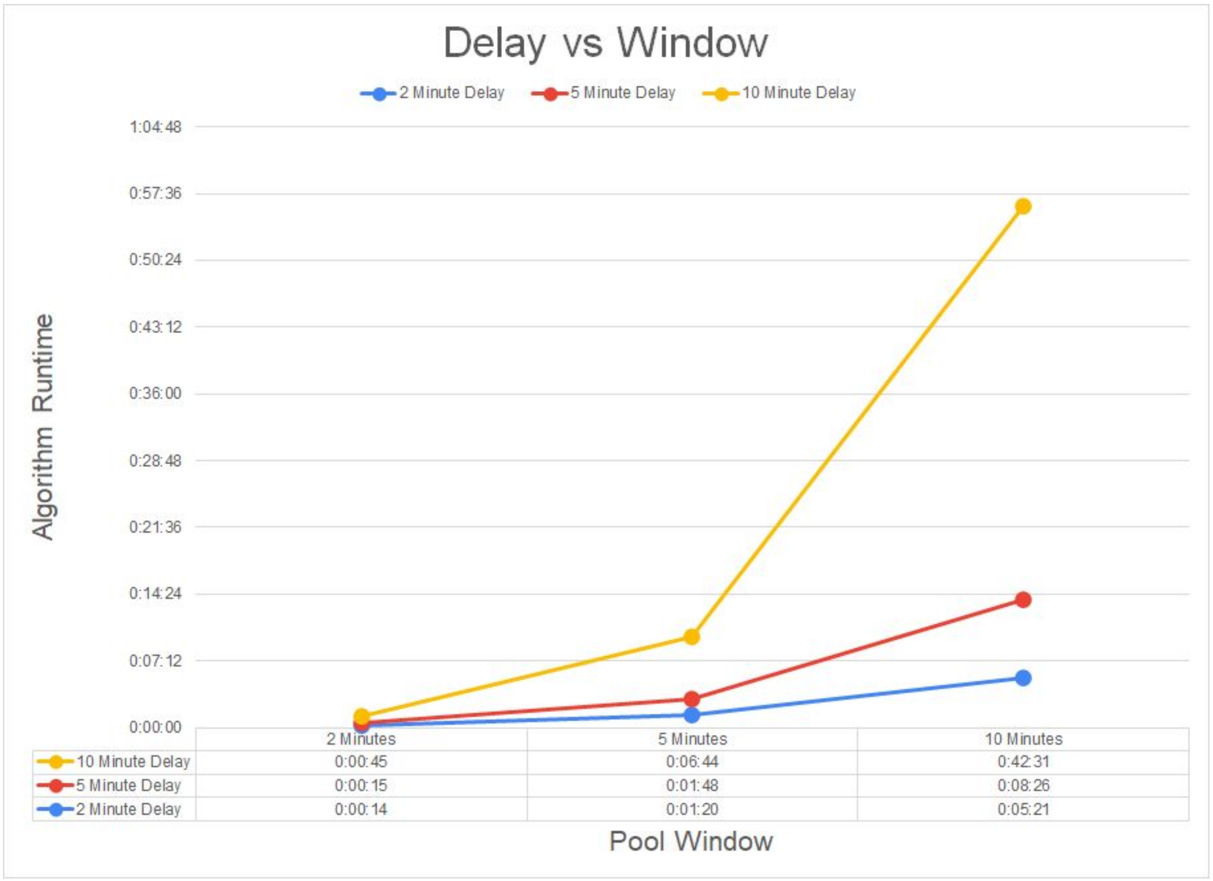
The values of the delay considered

Plot savings as a function of delay.

We considered the different times of 2, 5 and 10 minute delays and windows for the ride sharing. We took a sample size of a few hours to test the percentage of miles saved for each delay and window. Below is the results for the windows and delays with the time it took to run them:

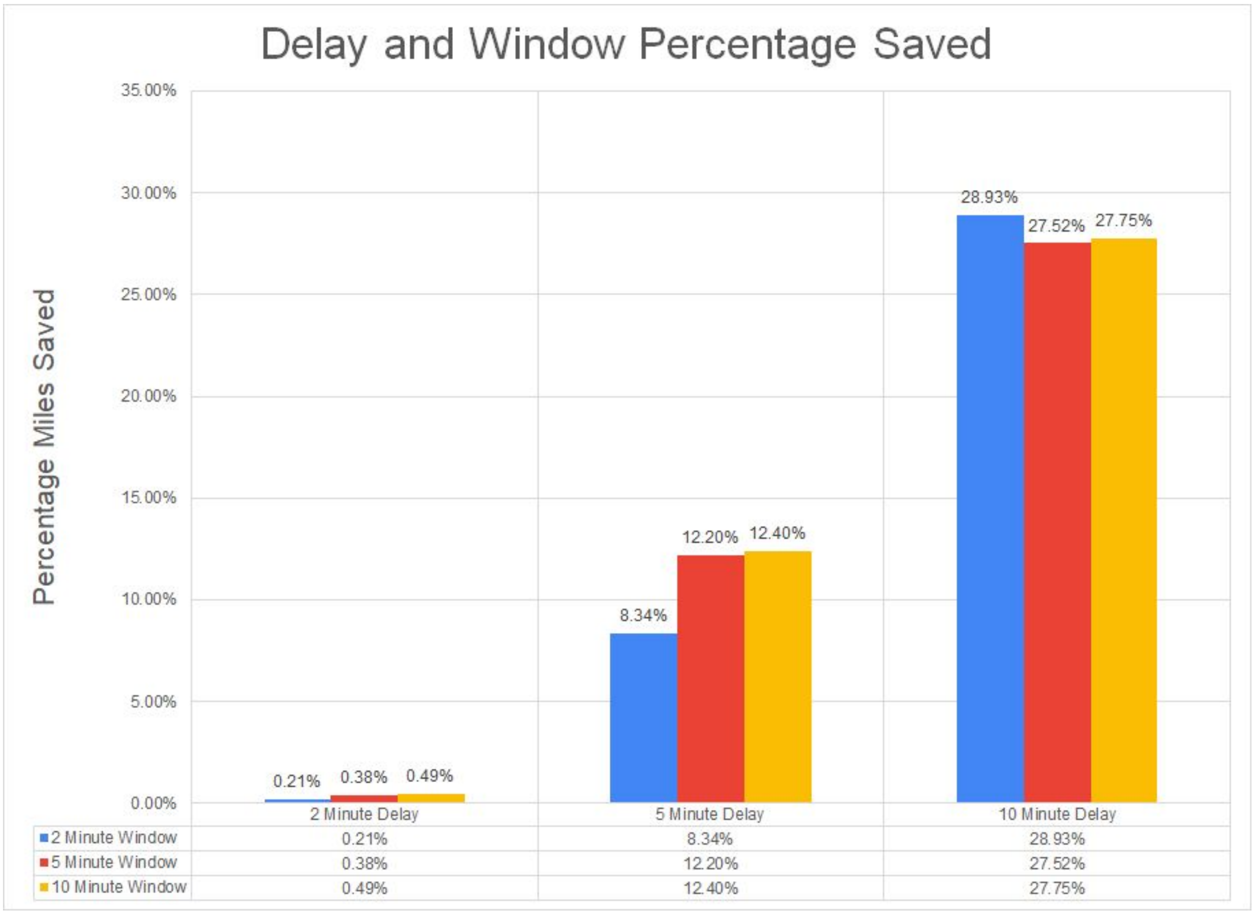
		2 min Delay	**5 min Delay**	**10 min Delay**
Window	2 mins	0 min 14 sec, 0.21%	0 min 15 sec, 8.34%	0 min 45 sec, 28.93%
Window	5 mins	1 min 20 sec, 0.38%	1 min 48 sec, 12.2%	6 min 44 sec, 27.52%
Window	10 mins	5 min 21 sec, 0.49%	8 min 26 sec, 12.4%	42 min 31 sec, 27.75%

Figure 10: *The effects of window and delay on runtime. As well as percent miles saved*



	2 Minutes	5 Minutes	10 Minutes
10 Minute Delay	0:00:45	0:06:44	0:42:31
5 Minute Delay	0:00:15	0:01:48	0:08:26
2 Minute Delay	0:00:14	0:01:20	0:05:21

Figure 11: *The effect of delay and window times on runtime*



	2 Minute Delay	5 Minute Delay	10 Minute Delay
🟦 2 Minute Window	0.21%	8.34%	28.93%
🟥 5 Minute Window	0.38%	12.20%	27.52%
🟨 10 Minute Window	0.49%	12.40%	27.75%

Figure 12: *The effect of delay and window time on percentage miles saved.*

Based on the results above, we chose a 5 minute window with a 5 minute delay because that section, we felt like as a group, had the best runtime to percentage saved ratio. A 5 minute window helped the calculations of the runtime of the data and a 5 minute delay makes sense because people don't want to wait for rides longer than 5 minutes.

Limitations and issues raised in presentations

Discuss what is missing in your project, and if/how you addressed questions raised in presentation

For our project, some of the limitations and issues we had was the runtime of the project and the installations of some of the libraries that we used. The major runtime issue we were encountering was the pooling algorithm, which ran in O(n^2) where n is the number of rides in a 5 minute window. This could take anywhere from a few seconds to up to 5 minutes to run and it was all dependent on the amount of rides that were included in a window. On average, it took about 3 hours and 45 minutes to run one days worth of data, so splitting up the workload between group members was a necessity. We quickly found out that you can run multiple instances of jupyter notebook at once, so being able to process a few weeks of data at one time was the only way we could get results for the entire year of 2015.

The other issues we ran into were the installation of dependencies for geopandas. We kept receiving errors when trying to pip install geopandas, and it was looking for external packages first. So we had to download the geopandas dependencies and install those manually, before we could pip install geopandas. Following the answer provided in this stackoverflow thread was extremely helpful to us.

During the presentation, professor Ouri asked us 3 questions:

- How did we retrieve our queries for the taxi?
- Did we combine the green and yellow data in our queries?
- Can he see results for all of NYC and not just for Manhattan?

For his first question, we were able to present him with our original query to get the information from our databases. We limited the results to 5 minute windows, and explained to him how that query ran very quickly due to our index we had created on the pickup/dropoff locations and times, and passenger count beforehand.

For the second question he had asked us, we were performing a union on the yellow and green taxi data to ensure every single ride in that 5 minute window was being retrieved.

Lastly, the professor asked us if he could see the results for all of NYC taxi data instead of just Manhattan. He noted that it was not a big deal since most of the rides would occur in just Manhattan, but he noted that he would be interested to see the data for the rest of NYC as well. In response to this, we were able to run one month of the full NYC taxi data for January in addition to every single month of just Manhattan.

Teamwork retrospective analysis

Discuss the specific contribution of each member; discuss teamwork.

Our team divided work evenly across all members, going so far to even have each member handle running a chunk of the dataset against the algorithm. Splitting up the dataset in this way allowed us to run the whole year of data in a time span much shorter than what it would have been on one machine. In terms of other work we set up several meetings to go over the project and write code. In these meetings all group members attended and we wrote the code collaboratively through Discord. Jupyter notebooks were used to write the code with one member taking the lead at one time and the rest commenting and coming up with solutions, this then would rotate to the next member as needed. The powerpoint and the final document were written in a similar fashion where a Discord meeting was set up and all members attended. These were able to be more interactive as all members were able to make changes at once allowing for a distribution of sections. These sections were edited and revised by all members until a consensus was reached on all material. For a more individual breakdown, Michael and Anthony were more responsible for the jupyter notebooks, Jia and Eric were more responsible for the presentation, and everyone worked equally on the report.

Appendix: Code and instructions to run it

Project Github Link

https://github.com/AnthonyVenturella/CS480_RideShare-Project

- Download Jupyter Notebook from the Anaconda Application
    - Anaconda | The World's Most Popular Data Science Platform
- Create a folder to store the following:
    - Jupyter file
    - Geojson file
- Install any dependencies that are needed
    - If there are any issues installing the geopandas dependencies for the Manhattan Notebook follow this link
    - Manhattan Notebooks
        - pymysql
        - pandas
        - numpy
        - networkx
        - haversine
        - matplotlib
        - geopandas
        - shapely
        - fiona
        - gmplot
        - folium

    - All of NYC Notebook
        - pymysql
        - nandas
        - numpy
        - networkx
        - Haversine

- Input MySQL connection settings in the 2nd jupyter cell
    - Host, port, user, passwd, and db
- Input location of manhattan geojson file in 3rd and 4th cell of Manhattan jupyter notebook