Share ride algorithm

Yu-Chien Huang

1 Propose a metric and/or algorithm to assess the potential efficiency of aggregating rides from many vehicles into one, given the available data. Make realistic assumptions and any necessary simplifications and state them.

The efficiency of aggregating ride can be evaluated by the "averaged saved distance", where the saved distance is the difference between the sum of the actual traveling distances of the rides that would be combined into a shared ride and the predicted traveling distance of the shared ride

$$\text{The efficiency of a shared ride} \quad := \frac{\text{sum of the actual distances of each rides} - \text{predicted distance of the shared ride}}{\text{the number of rides in the shared ride}} \tag{1}$$

[1]

In order to assess the potential efficiency, we first propose a reasonable share ride model.

1.1 Share ride model

The model composes three classes: Ride, Vehicle, gridMap. We create a Ride object for each good data point in yellow_tripdata_2016-06.csv in the Manhattan area [2]. All these Ride objects has the same gridMap object as an attribute, through which the Ride objects can interact with Vehicle objects that are created upon requested. The gridMap object keeps track of the dynamic states of all Vehicles objects that are created on the gridMap.

We require some reasonable conditions to be satisfied to share rides; it is important to consider carefully the flow of Vehicle objects and update the attributes accordingly. The three classes are defined in the file *new_classes.py*, we briefly describe them here:

- Ride: Each ride data point has a unique Id assigned in the order of the pick-up time. Other attributes of a Ride object includes the pick-up location, the drop-off location, the pick-up time, the drop-off time, the trip-distance, and the number of passengers. There are two methods in the Ride class: *request_time* and *request_vehicle*. *request_time* method sets the time that the passengers request a vehicle. We can be more sophisticate with this, but currently simply approximate it to be five minutes before the pick-up time. The *request_vehicle* method is called for each ride in the order of time/Id in the main function, which will find a vehicle for the ride on the gridMap.

[1]Since the exact routes are not given, we will use the shortest distances between a pair of points in our calculation in the (longitude, latitude) coordinate system, and map the calculated distances through a predictor to approximate real distances. The predictor is a simple linear regressor trained on the rides with features the 4th order polynomial features of the shortest distance between the pick-up and drop-off locations, and the labels the travel distances of the rides. Please see the file *distance_estimator.py* for the code.

[2]We disregard bad data points such as data points with zero or negative total durations or zero total travel distances.

- Vehicle: each Vehicle object is created with a unique Id and in *active*. The state of the Vehicle object is recorded in the dynamical attributes *current_time*, *current_position*, available *space* to take in more passengers, *moving_direction/velocity*. The history records are stored in the list *visited_list*[3] The dequeue object *to_visit_list* keeps track of the locations to visit where the pick-up location of a new added ride is left-appended and the drop-off location is appended. After a location is visited, the record will be left-popped to *visited_list*. The efficiency calculation will be based on these data. A vehicle will be out of commission if not requested before the last drop-off, and the *active* label will be set to False.

  Some class methods are described as follows:

  - *add_new_ride*: updates the attributes when a new ride is picked up
  - *arrival_time*: takes a new final destination as input, calculates the estimate arrival time
  - *distance*: a utility function that calculates the straight line distance between two given points
  - *driving_distance*: calculates the total lengths of the line segments connecting points in a given list
  - *duration*: a utility function that calculates the required time to travel between two given points
  - *end_of_time_statement*: calculates the predicted traveling distance of the share ride (share_distance) and the sum of the actual traveling distances of the rides that would be in the share ride in the simulation (no_share_distance); stores the data for the vehicle
  - *location_at_t*: takes a future time as input, returns the location of the vehicle at the time
  - *update_vehicle*: for a given time in the time line, left-pop *to_visit_list* and append *visited_list* accordingly

- gridMap: a Manhattan gridMap is shared by all the Ride objects. The gridMap is divided into roughly 10×10 grids. Other attributes of the gridMap include

  - *all_vehicles*, which is a dictionary that keeps the information of all actively running vehicles. The keys are the Ids of the vehicles and the values are the corresponding Vehicle objects.
  - *online_vehicle_map*, which is a dictionary with keys the grids of the gridMap and values a list of Vehicle Ids that are running in the grid.
  - *index*, which keeps track of the numbers of vehicles that have been created on the map, and indexes a new created vehicle.

  The class methods take care of the dynamics of the vehicles. Only under some reasonable pick-up conditions, an existing ride will be used to pick up another ride, if the conditions are not met, a new vehicle will be created. We search from the existing vehicles that are in the nearest nine grids around the request pick-up location, and the first pick-up ride will be the first to drop (the queue structure). More detailed descriptions of some methods are as follows:

  - *add_vehicle_to_map*: places a vehicle to a grid in the gridMap according to its position

[3]Each element is a tuple that records the action type ('i': pick-up, 't': turning point, 'f': drop-off), the time, the position, and the ride which the action is for.

- *find_vehicle*: finds a vehicle for a ride. If there is no existing vehicles that satisfy the pick-up conditions, creates a new Vehicle object for the ride
- *_from_existing_vehicle*: this method was called inside *find_vehicle*. The method collects candidate vehicles from the nearest nine grids around the pick-up location of the ride, and check if the pick-up conditions defined in *pick_up_conditions* are satisfied from the closest one.
- *grid_index*: calculates the grid index for a given position
- *pick_up_conditions*:
  1. The request time should not be before the time that the vehicle is created[4].
  2. The request time should not be after the time that the vehicle becomes inactive; i.e., after the vehicle has dropped all the rides.
  3. The vehicle should have enough space for the number of passengers in the ride.
  4. At the request time, the position where the request made should not have passed the vehicle; i.e., the angle between the vehicle moving direction should be within 90 degrees of the direction of the requested position.
  5. We should not add the ride to the vehicle if this will be too much detour (the last added ride will be the last to be dropped.)
  6. In contrast to the condition 5., this condition concerns about the detour problem regarding the vehicle's original next stop
- *remove_vehicle_from_map*: a vehicle will first be removed from its current grid to be place to a new grid.
- *update_gridMap*: this class method is called upon if *pick_up_conditions* are satisfied. Then we update the location of the to-be-shared vehicle with the turning point to pick up a new ride.

1.2 Setting model parameters

- *space* is the maximum number of passengers a vehicle can take, which is set to 6.
- *xy_minmax* is the four boundary points of Manhattan
- *harmonic_avg_speed* is determined the harmonic average of all ride's effective speeds, which is set to be the speed of vehicles
- *grid_length* is fixed by the distance that a vehicle travels in 15 minutes at *harmonic_avg_speed*
- There are four more parameters in *pick_up_conditions*: (1) angle_para = $\pi/2$ (2) detour_para= 0.3 (3) too_far_para= 1.3

2 Implement your proposed method and evaluate Manhattan's overall efficiency using yellow taxi data from the first full week (Monday-Sunday) in June 2016.

2.1 Time complexity

We have used both the list and queue data structures the same time to keep objects in time order while avoid searching or sorting. The visited states are in lists while the to visit states are in

[4]It is possible that a just created Vehicle object for a ride is within the nearest nine grids of a later ride of which the request time is however earlier than the pick-up time of the first ride (the time when the Vehicle objected is created).

queues, and the case that are currently being dealt is at the joint. All operations in the code are $O(1)$ except the operations to search for the nearest vehicle in the method *_from_existing_vehicle*. The sorting by nearest vehicle operation is $O(m \log m)$ in the number of vehicle m. If the total number of vehicles saturate eventually the overall time complexity is $O(n)$ in the number of rides n; if m and n are mapped through a non-constant sublinear function $m = f(n)$, the time complexity would be $O(n * f(n) \log f(n))$.

2.2 Overall efficiency

The overall efficiency of the first week in June 2016 is evaluated by the averaged saved distance of the rides that would be assigned to a shared ride (a ride that has more than one ride) in the our model. The total number of the rides that went into shared rides was 1,825,518, the actual distance sum of these rides was 3,055,066, and the predicted distance sum of all the shared rides was 2,608,271; hence the overall efficiency of the first week is 0.24475. The detailed results calculated from each day can be found in the table in next page. (Please see the calculation in the notebook *calculate_vehicles.ipynb*.)

	Mon	Tue	Wed	Thu	Fri	Sat	Sun
number of the rides that would be shared	262756	279397	284910	280146	217051	236288	264970
sum of the actual distances of the rides that would be shared	443206	469936	477712	477349	364733	383799	438331
sum of the predicted distance of all shared rides	377561	397812	405027	406298	312799	331891	376883
share ride efficiency	0.249834	0.258142	0.255116	0.253620	0.239271	0.219684	0.231905

Table 1

3 Based on your implementation in the previous question, use visualizations to show how efficiency varies with time and location.

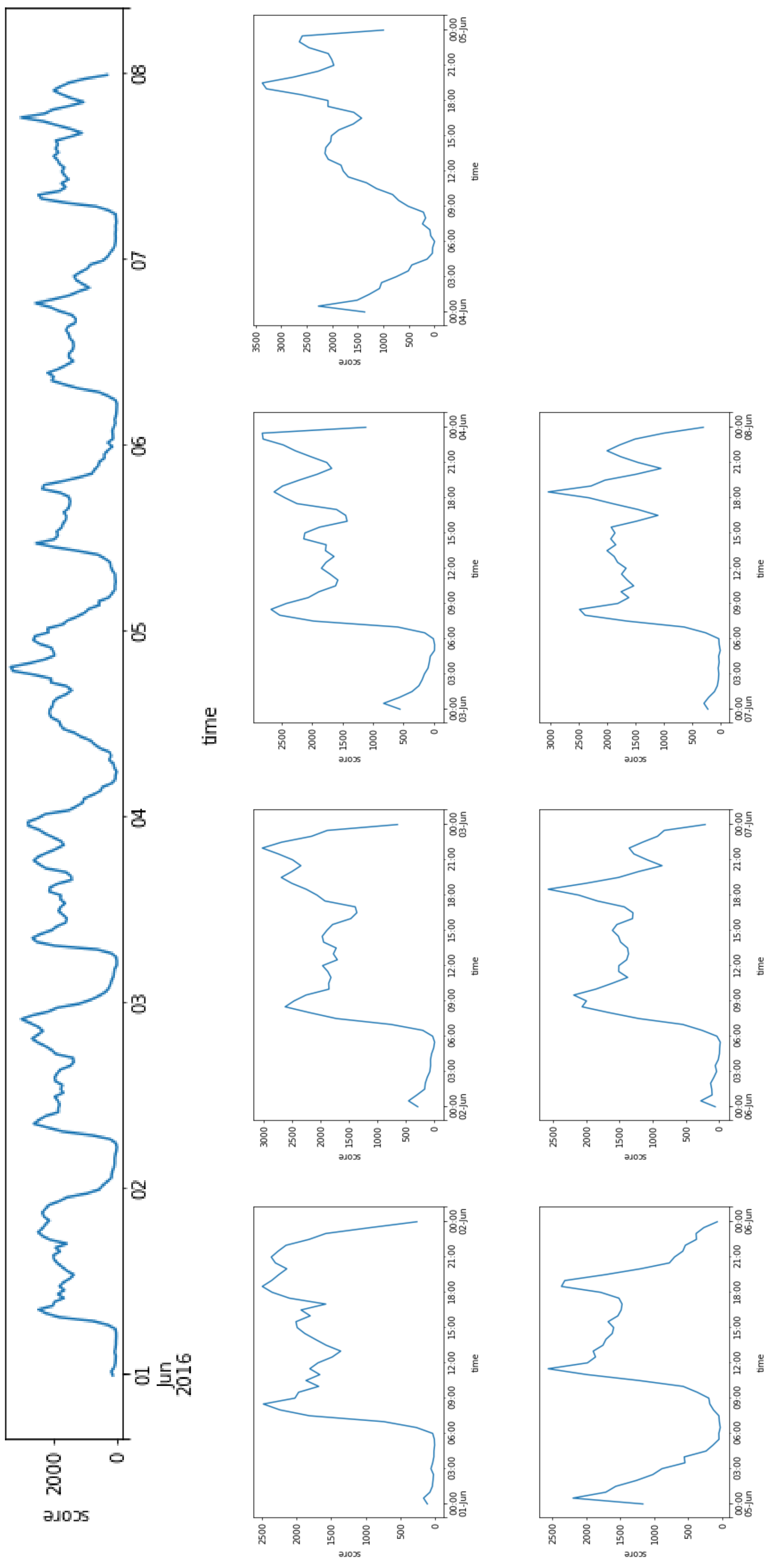
3.1 Time dependency

We discretize the time variable into half hours. The efficiency score for each half hours is determined as follows:

1. Collect the rides that were picked up by a shared ride in the period of the half hour

2. The efficiency score of the half hour is the value is the sum of the efficiency scores of all the shared rides associated with these rides

We plotted the efficiency score for the half hours in the whole week and the zoomed-in plots for each day as well. (Please see the notebook *time_dependency.ipynb* for generating the plots.)



score
time
01 Jun 2016
02
03
04
05
06
07
08
00:00 01-Jun
00:00 02-Jun
00:00 03-Jun
00:00 04-Jun
00:00 05-Jun
00:00 06-Jun
00:00 07-Jun
00:00 08-Jun

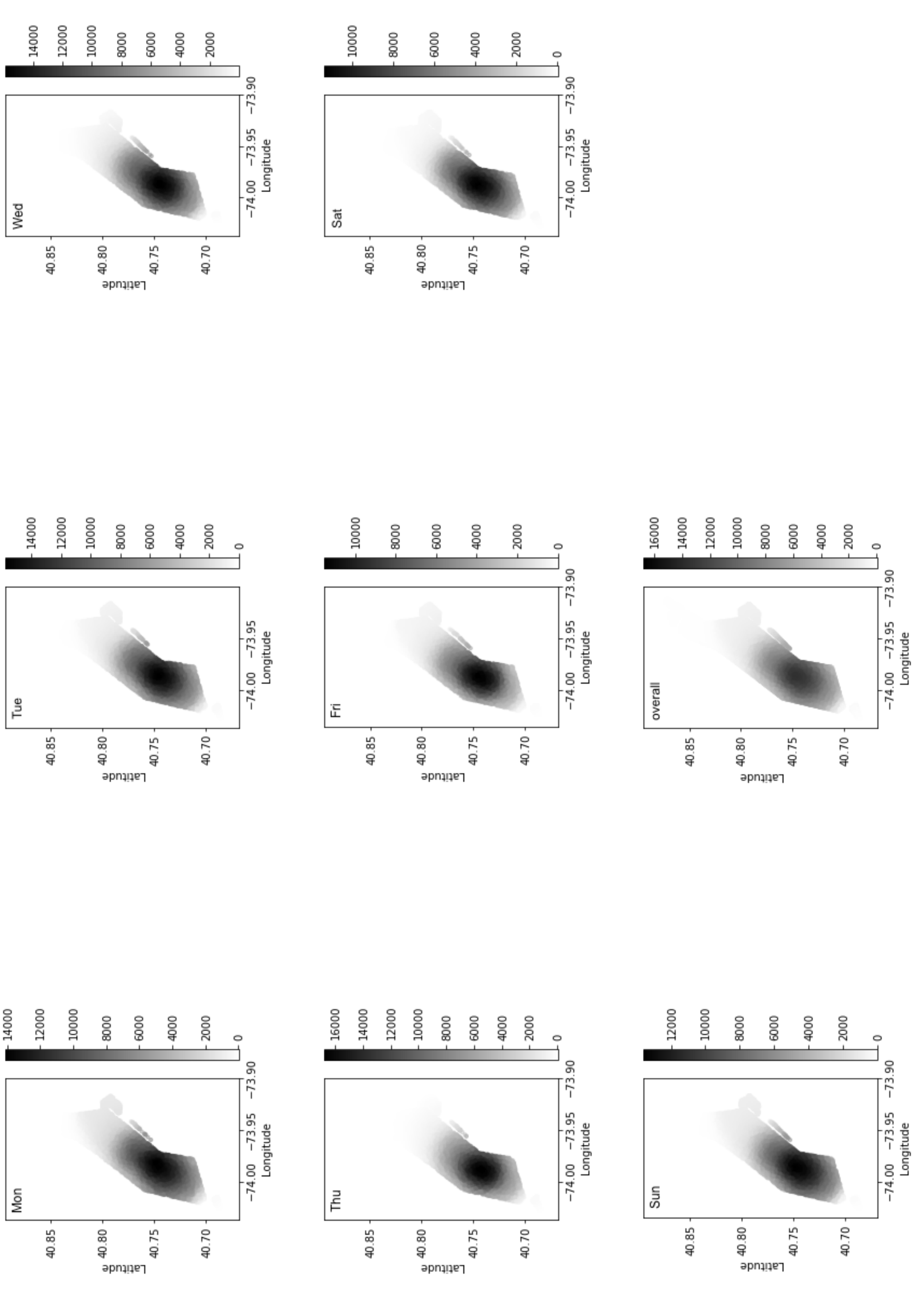
3.2 Location dependency

The way we determine the efficiency score for each point in the Manhattan area is as follows:

1. for each share ride, we find the smallest enclosing circle[5] of all points that the ride had visited (pick-up locations, drop-off locations, the turning points to pick up another ride)
2. for each circle area, we associate an efficiency value (calculated by equation (1)) of the shared ride to the circle area
3. for each point in the Manhattan area, we add up all the efficiency values of the circles that contain the point to be the efficiency value of the point

We drew 10,000 random points in the Manhattan area for each day, and plotted the efficiency scores of the 10,000 points calculated from the above procedure in the color map. The seven plots for each day are in the next page. We also include an overall plot from the points of all the seven plots. (Please see the notebook *location_dependency.ipynb* for generating the plots.)

[5]Smallest-circle problem Wiki page. We use a linear time solution code that can be found here.


Mon
Tue
Wed
Thu
Fri
Sat
Sun
overall
Latitude
Longitude
40.85
40.80
40.75
40.70
−74.00
−73.95
−73.90
0
2000
4000
6000
8000
10000
12000
14000
16000