Taxi in New York City: Generating Lucrative Passenger Pick-up Strategy

Julian Gao (julianyg)[1] and Qiujiang Jin (qiujiang)[2]

[1]Stanford University
julianyg@stanford.edu
[2]Stanford University
qiujiang@stanford.edu

I. Introduction

New York City, one of the most famous metropolis in the world, has an enormous collection of human activities at all time. Among those activities, one of the most significant symbols, the Yellow Cab, is representative of population and capital flow in the city. According to the NYC taxi cab fact book[1], there are over 485,000 taxi trips and 600,000 passengers per day during the year 2014. These numbers, however, are going down due to the impact from other car-hailing platforms such as Uber and Lyft. In order to maximize the profit for traditional Yellow Cab drivers and to compete with car-hailing companies, I propose a software framework to generate a set of optimized passenger pick-up strategies. The framework combines methods of machine learning, data mining, and Markov Decision Process (MDP). I show that my method when comparing with average hourly income from real data, performs much better. The framework I propose also applies to other user cases, which brings more social impacts other than helping NYC taxi drivers.

II. Related Work

There are many previous works on NYC taxi dataset. However, they are mainly focused on analysis and prediction aspects of data. Chriswhong visualizes the movement and earnings of a taxi in 24 hours[2]. Daulton *et al.* uses machine learning methods to predict pick-up density and drop-off locations[3]. Tipsy Taxi reveals the fraud behavior and performs traffic analysis, tip analysis for drivers[4]. There are studies on taxi passenger pick-up strategies, with binary behavior recommendation such as "hunting" or "waiting". Li *et al.*[5] and Tang *et al.* uses data with taxi occupancy label[6], which is not available on NYC dataset. Yuan *et al.* proposes and relies on parking places detection method, but targets on saving cruise time and maximizing trip distance[7].

III. Framework

A. Data Set

I use data directly from NYC Taxi & Limousine Commission (TLC) official website[8]. Starting from 2009, the TLC publishes the Yellow, Green, and FHV (For-Hire Vehicle) trip sheet data in CSV format. Each sheet contains trip records in a specific month, where each record is composed of the following information: VendorID, TPEP pick-up time, TPEP drop-off time, passenger count, trip distance, pick-up location, rate code ID, store and forward flag, drop-off location, payment type, fare amount, extra, MTA tax, improvement surcharge, tip amount, tolls amount, and total amount[9]. The TLC cannot guarantee or confirm the accuracy or completeness of data.

B. Software Pipeline

The framework is composed of two parts: policy (pick-up strategy) generation (1) and trip simulation (2). The policy generation part can be further divided into two stages: feature extraction (1.1), and MDP value iteration (1.2). Each stage is an independent module, where generated parameter files can be stored and distributed, for the convenience of reuse and computation, also serving sources for other data-related analysis.

IV. Methods

A. Assumptions & Simplifications

To simplify the problem, I make the following set of assumptions:

1) Locations are represented by grids in the form of tuples $((\text{lon}_W, \text{lon}_E), (\text{lat}_S, \text{lat}_N))$. Instead of using accurate GPS coordinates for locations, I choose to gridify the map region to reduce discreteness in location data. The grid size is controlled by the grid factor g, which is a trade-off between accuracy and complexity. The point coordinate of a grid is represented by that of its center point. One downside of using grids is that in baseline

policy, some non-reachable locations such as water area are included as states.
2) Region boundary $\mathcal{B}$ is reasonable. To reduce complexity and number of states, I use a boundary to exclude locations. If $\mathcal{B}$ is set too small, the generated policy may have not enough choices for the driver to choose from.
3) Taxi drivers are motivated by rewards, and will not pick-up passengers on the way to next target location. I later show in section C that the latter part, while seemingly counter-intuitive, is reasonable and has minor effects on output.

B. Data Mining & Feature Extraction

1) Data Pre-Processing: I deploy the Spark Python API (PySpark) for fast data manipulation, where huge files are stored in the form of Resilient Distributed Dataset (RDD). As mentioned in section III.A, the trip records in TLC dataset contain corrupted rows. Specifically, I find some trip distance, total amount, and location entries erroneous. Certain trip distances are a few thousand miles, and some payments are negative. The first step is to filter out the corrupted rows from input files. Next, I discard irrelevant entries and re-organize the key-value pairs. In order to adapt patterns and avoid over-generalizing the data, I decide to train on data from specific weekday and generate the policy for the same weekday. This takes account for different passenger behavior on different weekdays. The final data are stored as key-value pairs, where the key is a grid-hour tuple (pick-up location, time), and the value contains trip distance, trip duration, average speed, drop-off location, and total pay amount.

2) Parameters: To evaluate the best pick-up location, I take account of the following grid parameters: pick-up probability p, payment amount distribution μ_p, σ_p, trip distance distribution starting from the grid μ_s, σ_s, trip time distribution starting from the grid μ_t, σ_t, and a drop-off location probability map. I now explain each parameter in details.

1) Pick-up probability p. The passenger pick-up process within a grid g can be modeled by a standard Poisson process, where λ is the average waiting time for a pick-up in g. The average waiting time is calculated by the formula $\lambda_{(g,t)} = \frac{1}{D}\sum_d \frac{60}{\mathcal{P}_{(g,d,t)}}$, where D is the total number of weekday d in the dataset, and $\mathcal{P}_{(g,d,t)}$ is the total number of pick-up's in grid g during hour t of day d. Now that I have $\lambda_{(g,t)}$, I need to know the driver's waiting time in grid g. I assume the driver cruises in g until he meets any passenger. The cruising time t_c can be calculated by the size of g and cruising speed v_c. While the size l_g is easily calculated by using haversine formulas with the GPS coordinates, the cruising speed v_c is not included in data. Hence I introduce the concept of **congestion factor** α to infer v_c from the average trip speed v_t. The congestion factor α is calculated by the **workload** ω_g of the grid, which maps an exponential decay of ω_g to a more smooth scaling of taxi speed v_g. The workload ω_g is defined as

$$\frac{\mathcal{P}_{(g,t)} + \mathcal{D}_{(g,t)}}{\sum_g (\mathcal{P}_{(g,t)} + \mathcal{D}_{(g,t)})} \tag{1}$$

, the number of pick-up and drop-off activities at certain hour t in grid g divided by that of the entire map. Here I assume the taxi activity has a strong positive correlation with the traffic congestion, since more activities indicate a hotspot, thus more traffic. The congestion factor α is defined as a sigmoid:

$$\alpha_g = 2(1 - \frac{1}{1 + e^{-K(\omega_g - \mu_\omega)}}) \tag{2}$$

$$v_g = \alpha_g v_t \tag{3}$$

The congestion factor is scaled by 2 because I assume the average speed at any grid is within amplitude of 2 of the average trip speed. The sigmoid function is intuitive to use, due to the critical points on its curve: on an empty street, the speed is harder to increase; and inside a jam, the speed is unlikely to decrease. The steepness factor K is tuned to accommodate another important feature: across the set of grids $\mathcal{G}$ passed on a trip, the average of α calibrated speed v_g should still average to trip average v_t. This can be generalized to the continuous case such that

$$\int \alpha v_t \cdot p(\alpha, s) d\alpha = v_t \tag{4}$$

where $p(\alpha, s)$ is the portion of grids with congestion factor α in a trip with distance s. Equation (4) can be further simplified to

$$\int \alpha p(\alpha, s) d\alpha = 1 \tag{5}$$

To prove this case, first I need to find the distribution function $p(\alpha, s)$. I plot the histogram of Number of Occurrence *vs.* ω at Tuesday 12 am on a small but representative dataset in figure 1. This is representative in all (day, hour) combinations. Apparently $p(\alpha, s)$ follows an exponential distribution, where

$$p(\alpha, s) = \kappa \frac{L}{s} e^{-\kappa \frac{L}{s} \alpha} \tag{6}$$

Here $\frac{L}{s}$ is the scale factor for κ, where L is the constant number of map size. This

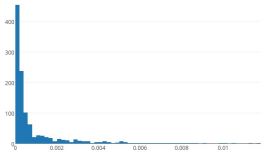

Fig. 1. Tuesday 12 am Workload Distribution

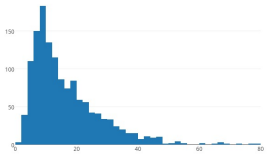

Fig. 2. 5 pm Little Italy SOHO Trip Time Distribution

can be interpreted as the longer distance the driver traveled, the more he/she is likely to see the real distribution. With a small value of s (short trip distance), by common sense, the driver is more likely to see grids with smaller ω, equivalent to a steeper exponential probability curve after normalization. Now if I substitute $\kappa\frac{L}{s}$ as γ into (5), I get

$$\int_0^1 (1 - \frac{1}{1+e^{-K(\omega-\mu_\omega)}})\gamma e^{-\gamma\omega} d\omega = \frac{1}{2} \tag{7}$$

For this equation to hold, I tune the steepness factor K empirically as $\frac{\mu_\omega}{2\sigma_\omega^2}$. This setup is later proved to work out well. With the cruise time calculated from $t_c = \frac{v_t}{\alpha v_g}$ and average waiting time λ, I sample from the process and calculate the pick-up probability:

$$p = \sum_{i=0}^{t_c} \frac{\lambda^i e^{-\lambda}}{i!} \tag{8}$$

2) Distribution mean, variance. I plot the histogram of trip distance, trip duration, and total pay amount starting from several POI's at different time on Tuesday in figure 2-4.

Apparently, from the figures, I notice all three distributions are skewed normal. The means and variances of theses distributions are thus effective in evaluating the reward of a grid. Given the numbers, for the specific week day, I define the following reward function for a (grid, hour) key:

$$R(g,t) = p\frac{(\mu_p - \frac{\sigma_p}{2})^2}{(\mu_t - \frac{\sigma_t}{2})(\mu_s - \frac{\sigma_s}{2})} \tag{9}$$

I will discuss later their usage in path simulation.

3) Drop-off location probability map. This is crucial data for the MDP process and path simulation. Given location-time pair (g,t), I store the mapping

$$\mathcal{M} : (g,t,g') \to \{\rho_{g'}\} \tag{10}$$

for all g' in the drop-off location set $\mathcal{D}_{(g,t)}$, where ρ is the probability of drop-off at g'. Note that $\sum_{g'} \rho_{g'} = 1$.

With above parameters described, I visualize a few pick-up probability maps of NYC, generated from a 15-month dataset. I also plot the drop-off location map from certain pick-up locations. See figure 5-9.

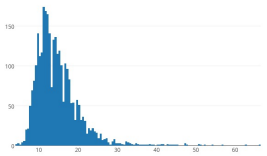

Fig. 3. 6 pm Lincoln Tunnel Pay Distribution

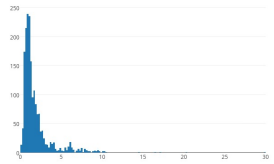

Fig. 4. 11 am Sutton Place Trip Distance Distribution

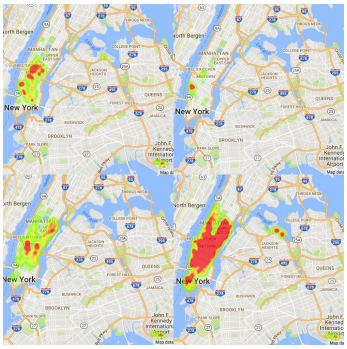

Fig. 5. From Top-Left to Bottom-Right: 12 am, 3 am, 6 am, 9 am

C. Markov Decision Process

Now I formally define the MDP for the passenger pick-up problem.

States: $\mathcal{S} = \{(g,t) : g \in \mathcal{B} \wedge t \in (T_s, T_e)\}$, where T_s, T_e are start and end time (driver shift).

$\mathbf{s_{start}} = (g_s, T_s)$ where g_s is starting location.

Actions$((g,t))$ =

$$\{(g',t') : t' = t + t_s \wedge g' \in \mathcal{N}(g) \cup \mathcal{H}\} \tag{11}$$

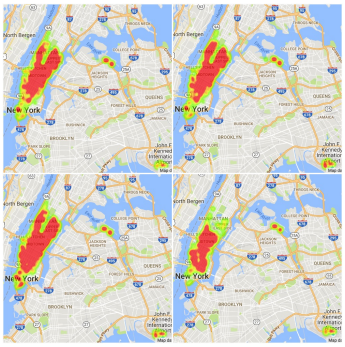

Fig. 6. From Top-Left to Bottom-Right: 12 pm, 3 pm, 6 pm, 11 pm

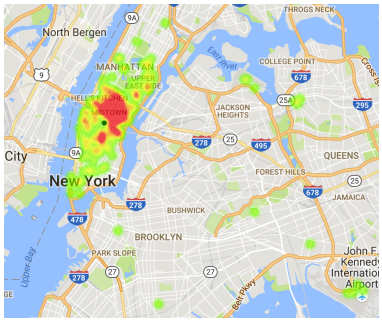

Fig. 7. 5 pm Drop-off From Empire State Building

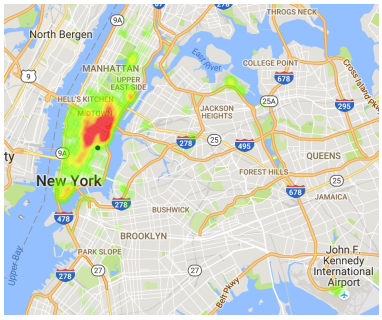

Fig. 8. 9 pm Drop-off From Mt. Sinai Hospital

Fig. 9. 11am Drop-off From Chanel Shop

where t_s is the trip duration, $\mathcal{N}(g)$ is the neighbors of g (including itself), and $\mathcal{H}$ is the set of hotspots on the map.

Transition$((g,t),a,(g',t'))$=

$$1\{g' \in \mathcal{N}(g)\}(1-p_g) \tag{12}$$

$$+1\{g' \notin \mathcal{N}(g)\}p_g \cdot \mathcal{M}_{(g,t,g')} \tag{13}$$

which can be viewed as two parts: if the taxi does not pick up a passenger in current location, then it can arrive in the neighbors; otherwise it can only arrive at location g' if the passenger also goes there.

Reward$((g,t),a,(g',t')) = \frac{R(g,t)}{t'-t}$

This is simply dividing reward function (9) by the travel time towards g'. In current implementation δ_t is calculated by $\frac{g'-g}{v} + t_c$. The result can be more accurate by applying Bresenham's Algorithm, which is discussed in section VI.

IsEnd$((g,t)) = 1\{t > T_e\}$. Exceeds time limit.

γ = 1.

Now the logic behind the scene. There are many assumptions to make to solve the problem: if I assume the driver to pick up whenever he/she sees passengers on the way, then I can only allow Actions$((g,t))$ to be $\mathcal{N}$, otherwise there is no guarantee that the driver will ever arrive at any chosen g'. However, if I only allow Actions$((g,t))$ to be $\mathcal{N}$, then if the driver is in the middle of nowhere (unseen states in dataset), then he/she will get bogged.

On the other hand, if I assume the driver only goes to chosen target g' without taking any passengers midway, then I can allow Actions$((g,t))$ to be arbitrary $g' \in \mathcal{B}$. This setting explores the entire map with every possible situation, which finds the global optimum of the problem. However, this is computationally unacceptable. Instead, I decide to take a step back, and choose to search for local optimum: let Action$((g,t))$ to be $\mathcal{N} \cup \mathcal{H}$, a combination of neighbors and hotspots. This prevents the driver from wandering in nowhere and also stops him/her from over-exploiting the hotspots. This is an analogy of the Exploration-Exploitation problem, where ϵ stands for the size of $\mathcal{H}$. My assumption of not picking up midway also works perfectly here: if the driver is heading to the hotspots, the g's on the path will not likely have high pick-up probabilities, otherwise, they are hotspots themselves.

D. Policy Generation

1) Learned Policy: I use value iteration method to generate the learned policy for MDP states. For each state s, compute the utility value as

$$U_{\text{opt}}^{(t)} \rightarrow \max_{a \in \text{Actions}(s)} \sum_{s'} T(s,a,s')[R(s,a,s') + U_{\text{opt}}^{(t-1)}(s')] \tag{14}$$

Different from what I stated in the formulation of MDP above, in the actual implementation I

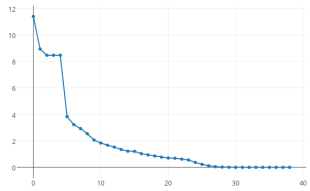

Fig. 10. Value Iteration Learning Curve

choose states as the set of states appeared in dataset keys. This is to reduce unnecessary computation and expedite the iteration. If the successor (g', t') from (g, t) is unseen (new t' from time elapse, or new g' from wandering into neighbors), then I assume average t_c, v, and $p = 0.0$, since not recorded means there is no previous pick-up event. The probability sums to 1 as follows:

$$p + (1 - p) \sum_{g'} \rho_{g'} = 1 \tag{15}$$

Figure 10 is the learning curve for value iteration.

2) Baseline Policy: An obvious baseline policy is to run randomly on $\mathcal{N}$. One disadvantage mentioned before is that the driver may run into unreachable spaces such as the middle of Central Park, or Hudson Bay. Using coordinates of POI's can solve this problem.

3) Oracle Policy: The main obstacle for achieving an oracle policy is that the TLC dataset is highly anonymous, with all license and Medallion ID erased from records, thus I cannot follow on individual taxis to directly pick out the route with maximum net profit. I come up with a dynamic programming get-around. Different from the baseline and learned policies, the oracle takes in the initial location g_i to generate an optimal route. I first process the original RDD to reverse link all states, so that a key-value pair has form of $s' \rightarrow (s, m(s, s'))$, where $m(s, s')$ is the utility earned by going from $s = (g, t)$ to $s' = (g', t')$, evaluated by the Reward function. The base case is initializing all utilities to be zero: $u_s = 0$ for all $s \in \mathcal{S}$. Next, for all states, I define the transition function as

$$u_{s'} = \max(u_s + m(s, s'), m(s_i, s')) \tag{16}$$

where s_i is the initial state. Note that all unseen states are assigned with $u = 0$. Backtracking the utility map, I get the oracle policy.

V. Results

A. Policy Analysis

I run the framework on the same small representative dataset, run generated policies on same location and time settings (Tuesday 12:01 am, Empire State Building, 80 minutes shift) for 30 iterations, and use the average reward to evaluate the correctness of my methods. Below is the table comparing results from three policies (Note: Performance is the ratio between profit and depreciation. Depreciation is the measurement of fuel cost).

Policy	Profit	Depreciation	Performance
Baseline	51.93	14.94	3.48
Learned	74.99	19.55	3.84
Oracle	79.97	20.66	3.87

The result clearly shows the performance rise. Both learned and oracle policy performs much better than the baseline policy. I choose a hotspot as starting point, so that the driver can always move to another location at the first trip. The baseline policy at the new location starts random walk, and is harder to find passengers at midnight if the drop-off location is non-ideal. The fact that passengers from Empire State Building typically go to other hotspots reduces this effect, and the baseline policy still gains a decent amount of profit. In other cases it performs much worse.

The oracle slightly outperforms the learned policy, because in real world, drivers are not always making optimal decisions. Besides, the simulation samples the drop-off locations from a multinomial distribution, which cannot relive the real situation perfectly.

B. Visualization of Simulated Path

In order to better extrapolate the feasibility and correctness of my framework, I run the generated policies on my simulator and visualize some representative routes in figure 11-13, starting from Tuesday 12:01 am from Empire State Building, driving for 80 minutes. The full visualization part is included in the code repository, with drop pins marking earned money, and time. In simulation phase, I sample the multinomial drop-off locations from $\mathcal{M}$. Note that the profit values are higher than the actual case of average 50$, due to the slightly large fare rate in the simulation.

VI. Future Works

As mentioned previously, the gridify process is a bold assumption I make, and migrate to POI GPS coordinates can make the framework more accurate. In evaluating travel time from current location to target location, I simply divide the straight line distance by the average speed, which is not ideal. The Bresenham's Algorithm determines the points of an n-dimensional raster that should be selected in order to form a close approximation to a straight line between two points[10], from where I can make use of congestion factors along the

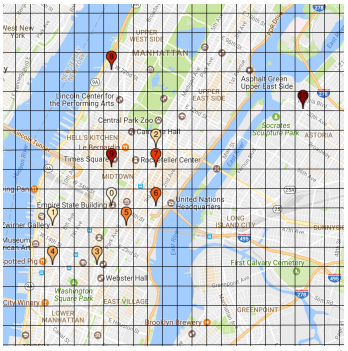

Fig. 11. Oracle Route. Earned 104.01$

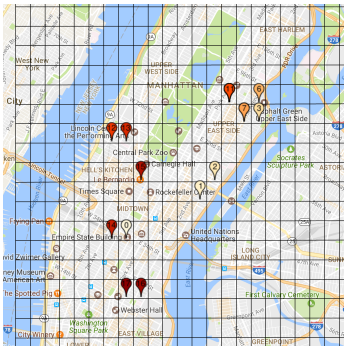

Fig. 12. Learned Route. Earned 83.80$

path and get a more accurate estimation of travel time between grids. As a side mark, I will run on larger dataset for more comprehensive policies, with extra computational resources such as AWS. Code available on GitHub repository:

https://github.com/JulianYG/221finalproject

REFERENCES

[1] N. T. . L. Commission, "2014 taxicab fact book," 2014.

[2] C. Wong, "A day in the life, a data visualization of a single nyc taxi over 24 hours," 2015.

[3] S. R. Samuel Daulton and T. Kindt, "Nyc taxi data prediction," 2015.

[4] lucianoiscool, "Tipsy taxi," 2014.

[5] B. Li, D. Zhang, L. Sun, C. Chen, S. Li, G. Qi, and Q. Yang, "Hunting or waiting? discovering passenger-finding strategies from a large-scale real-world taxi dataset," *IEEE Xplore 10.1109/PERCOMW.2011.5766967*, 2011.

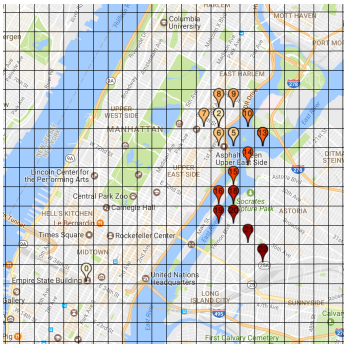

Fig. 13. Baseline Route. Earned 25.09$. Gridify Effect drives the cab into water.

[6] H. Tang, M. Kerber, Q. Huang, and L. Guibas, "Locating lucrative passengers for taxicab drivers," *ACM SIGSPATIAL 10.1145/2525314.2525471*, 2013.

[7] J. Yuan, Y. Zheng, L. Zhang, X. Xie, and G. Sun, "Where to find my next passenger?," *UbiComp 10.1145/2030112.2030128*, 2011.

[8] N. T. . L. Commission

[9] N. T. . L. Commission, "Data dictionary - yellow taxi trip records," 2015.

[10] Wikipedia, "Bresenham's line algorithm,"