

VICTORIA UNIVERSITY OF WELLINGTON  
*Te Whare Wananga o te Upoko o te Ika a Maui*



School of Engineering and Computer Science  
*Te Kura Mātai Pūkaha, Pūrorohiko*

PO Box 600  
Wellington  
New Zealand

Tel: +64 4 463 5341  
Fax: +64 4 463 5045  
Internet: [office@ecs.vuw.ac.nz](mailto:office@ecs.vuw.ac.nz)

**COMP440 Assignment 3**  
**Kaggle Competition - New York City Taxi Fare Prediction**

Han Han (300474699)

Supervisor(s): Will Browne

# Part 1: Exploring and understanding the Taxi data

## 1. Reading raw data

train.csv contains more than 55 million rows of data. To read this file into one single DataFrame, several methods have been tried. Most of them crashed the computer or processed for too long time. Two practical methods are selected in this report.

The first one used the `read_csv` method in pandas library. The parameter `rows` can be input by user to define how many lines to read from the .csv file.

```
df_train = pd.read_csv('train.csv', nrows = rows,
                      parse_dates=["pickup_datetime"])
```

This function works well when the input is less than around 2 to 3 million. Normally it can load the first 2 million rows into memory within 10 minutes, which takes long time but is acceptable. The problem is once the input is much larger than 2 million rows, the reading will end up with “insufficient memory” warning.

Another method which makes reading all the 55 million rows feasible was published by szelee on <https://www.kaggle.com/szelee/how-to-import-a-csv-file-of-55-million-rows>.

```
df_train = dd.read_csv('train.csv').compute()
df_train['pickup_datetime'] = df_train['pickup_datetime'].str.slice(0,16)
df_train['pickup_datetime'] = pd.to_datetime(df_train['pickup_datetime'],
                                           utc = True,
                                           format = '%Y-%m-%d %H:%M')
df_train.index = pd.RangeIndex(start = 0, stop = len(df_train_i))
```

Dask library was imported as `dd` for this application. The `compute()` method following the `dd.read_csv('train.csv')` creates a pandas DataFrame.

The reason why this method can read all 55 million rows of data is explained in below website: <http://pythondata.com/dask-large-csv-python/>

*“With Dask and its dataframe construct, you set up the dataframe must like you would in pandas but rather than loading the data into pandas, this approach keeps the dataframe as a sort of ‘pointer’ to the data file and doesn’t load anything until you specifically tell it to do so.”*

Also, comparing to using `parse_dates`, the 2<sup>nd</sup> and 3<sup>rd</sup> lines in `read_full()` function above work much faster.

`RangeIndex` method here is used to rearrange the index from 0 to the length; otherwise, the index only repeats from 0 to a number around 10000. This may be because the data was read into memory and processed by chunks.

`dd.read_csv` method can also read the .csv file partially by following the `.head()` method as below:

```
df_train = dd.read_csv('train.csv').compute().head(10000)
```

But comparing to `pd.read_csv`, this method takes longer time.

Several data reading experiments were done with different combinations of parameters. The result is as below:

Library	parse_dates	Lines	Result
Pandas	use	2 million	3 min 43 s
Pandas	not use	2 million	10 s
Dask	not use	2 million	5 min 28 s
Dask	not use	55 million	7 min 45 s

So, when reading only thousands of rows of raw data for test run, `pd.read_csv` (without using `parse_dates` keyword) is time-saving; `dd.read_csv` is good at reading much larger .csv file.

If printing the last 5 rows of the DataFrame using `.tail()` method, the result shows the raw data has 55,423,856 rows.

```

key fare_amount ... dropoff_latitude passenger_count
55423851 2014-03-15 03:28:00.00000070 14.0 ... 40.762555 1
55423852 2009-03-24 20:46:20.0000002 4.2 ... 40.773959 1
55423853 2011-04-02 22:04:24.0000004 14.1 ... 40.797342 1
55423854 2011-10-26 05:57:51.0000002 28.9 ... 40.773963 1
55423855 2014-12-12 11:33:00.00000015 7.5 ... 40.783313 1

```

## 2. Data patterns exploration and Visualization

### 2.1 Data types

`.info()` method was used to print information about the DataFrame, especially the data types of each column.

```
Data columns (total 8 columns):
key                object
fare_amount        float64
pickup_datetime    datetime64[ns, UTC]
pickup_longitude   float64
pickup_latitude    float64
dropoff_longitude  float64
dropoff_latitude   float64
passenger_count    int64
dtypes: datetime64[ns, UTC](1), float64(5), int64(1), object(1)
```

There are 7 features in raw data (except for the key), all are stored in 64-bit data type. To minimise the processing time, some of them can be converted to 32-bit. However, in later stage of experiment, it has been proved that converting float64 to float32 led to the loss of accuracy of longitude and latitude data, which is unacceptable; also, when converting the fare\_amount from 64-bit to 32-bit, the accuracy was also decreased. Many data were ending with x.xx99999 or x.xx00001 after the conversion, which increases the difficulty in judging whether the raw data fulfil any pattern or condition and further cleansing, processing and analysing.

### 2.2 Missing Data

`.isnull().sum()` method was used to summarize the missing data information about the training dataset.

```
key                0
fare_amount        0
pickup_datetime    0
pickup_longitude   0
pickup_latitude    0
dropoff_longitude  376
dropoff_latitude   376
passenger_count    0
dtype: int64
```

There are 376 missing data instances. All those cases, drop-off locations are missing. Those rows of data have to be removed before further data analysing.

### 2.3 Statistical Information

`.describe()` method was used to generate descriptive statistical information of each attributes in the dataset.

#### 2.3.1 Training Set

	fare_amount	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	passenger_count
count	5.542386e+07	5.542386e+07	5.542386e+07	5.542348e+07	5.542386e+07	5.542386e+07
mean	1.134505e+01	-7.250968e+01	3.991979e+01	-7.251121e+01	3.992068e+01	1.685380e+00
std	2.071083e+01	1.284888e+01	9.642353e+00	1.278220e+01	9.633346e+00	1.327664e+00
min	-3.000000e+02	-3.442060e+03	-3.492264e+03	-3.442025e+03	-3.547887e+03	0.000000e+00
25%	6.000000e+00	-7.399207e+01	4.073493e+01	-7.399140e+01	4.073403e+01	1.000000e+00
50%	8.500000e+00	-7.398180e+01	4.075265e+01	-7.398015e+01	4.075316e+01	1.000000e+00
75%	1.250000e+01	-7.396708e+01	4.076713e+01	-7.396367e+01	4.076810e+01	2.000000e+00
max	9.396336e+04	3.457626e+03	3.408790e+03	3.457622e+03	3.537133e+03	2.080000e+02

We can see the maximum and minimum value for each column are out of range of common sense and business understanding, such as negative fare amounts and even a case with a fare amount around \$90,000, passenger count of 208, longitude and latitude of position on the other side of the earth. So, a data cleansing process is a must in a later stage.

### 2.3.2 Test Set

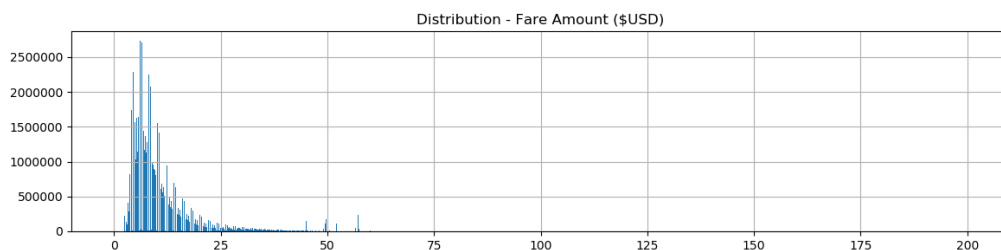
	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	passenger_count
count	9914.000000	9914.000000	9914.000000	9914.000000	9914.000000
mean	-73.974722	40.751041	-73.973657	40.751743	1.671273
std	0.042774	0.033541	0.039072	0.035435	1.278747
min	-74.252193	40.573143	-74.263242	40.568973	1.000000
25%	-73.992501	40.736125	-73.991247	40.735254	1.000000
50%	-73.982326	40.753051	-73.980015	40.754065	1.000000
75%	-73.968013	40.767113	-73.964059	40.768757	2.000000
max	-72.986532	41.709555	-72.990963	41.696683	6.000000

Form the printing, we can see there are 9914 cases to be predicted. The passenger count range is from 1 to 6; latitude range (including both longitude and latitude) is from 40.56 to 41.71; longitude (including both longitude and latitude) is from -74.27 to -73. This may give us a rough guideline in pre-processing the data in training dataset.

## 2.4 Data Distribution

### 2.4.1 Training Set

The distribution of each feature has been plotted below.

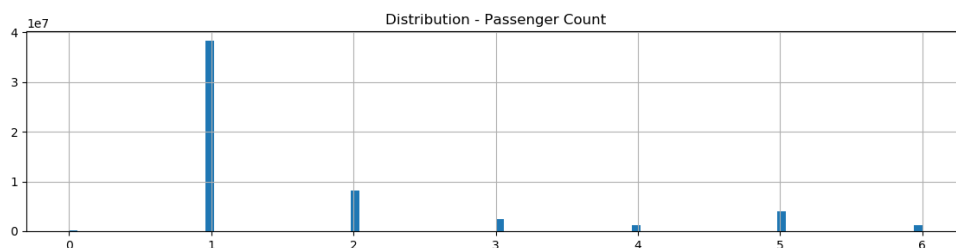


From the fare amount distribution, we can see that most of the trips do not cost more than \$25, and there are some discrete popular fares in \$5 to \$25 range (high spikes in histogram), which may indicate some fixed fares from location A to location B in NYC. There are also some bars in a \$10 range around \$50, which may be the flat fare from Manhattan to JFK airport (\$52 + tolls + surcharges).

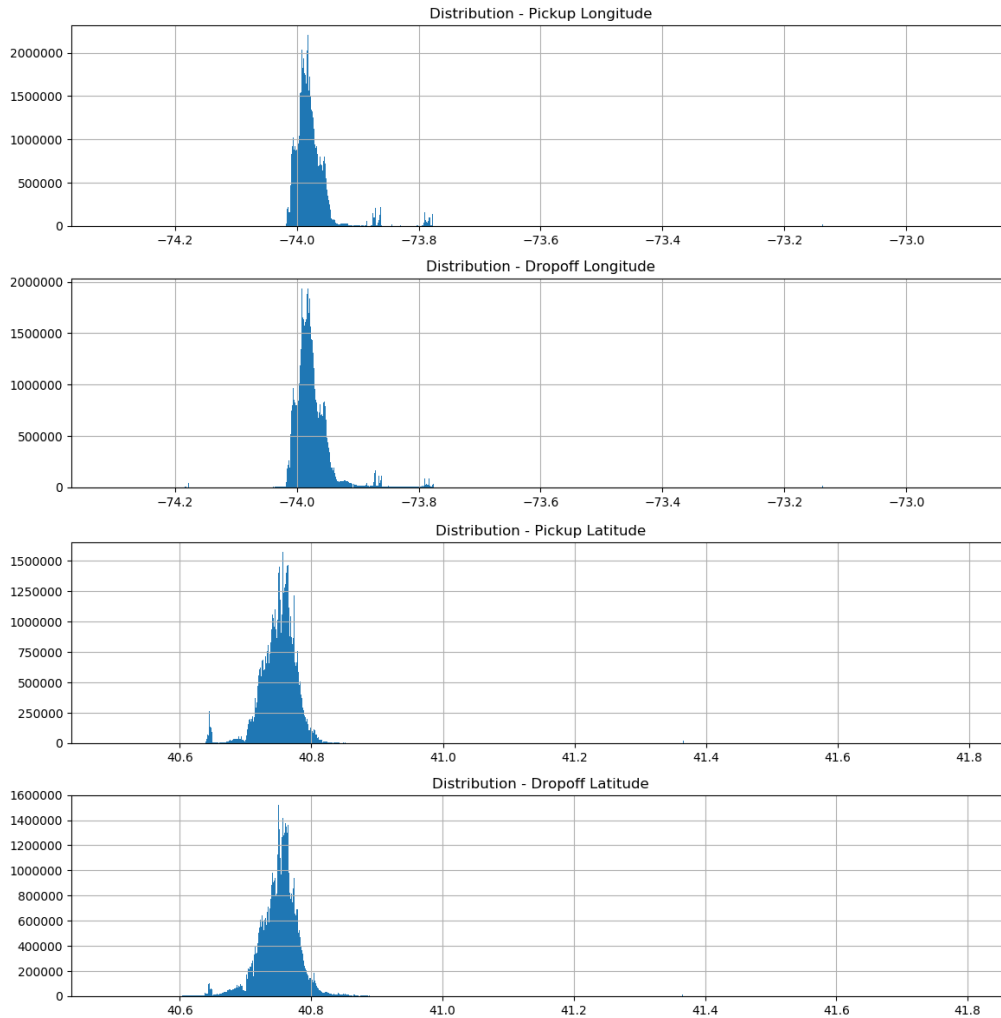
*“To/From JFK and any location in Manhattan:*

*This is a flat fare of \$52 plus tolls, the 50-cent MTA State Surcharge, the 30-cent Improvement Surcharge, and \$4.50 rush hour surcharge (4 PM to 8 PM weekdays, excluding legal holidays).”*

source: [http://www.nyc.gov/html/tlc/html/passenger/taxicab\\_rate.shtml](http://www.nyc.gov/html/tlc/html/passenger/taxicab_rate.shtml)



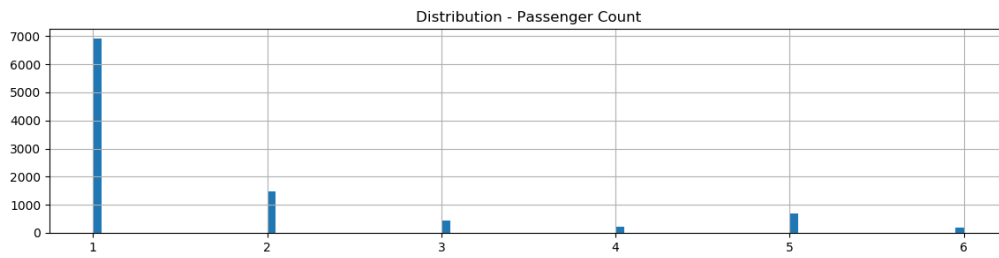
The distribution of passenger count shows that most of the trips (almost 40 million trips) were taken by only 1 passenger. There are also millions of trips with 2, 3, 4, 5, 6 passengers. Also, because of its high amount, the passenger count with 0 passenger should not be overlooked directly, even though it may be due to wrong input of the taxi drivers or it is just for return trip or cargo delivery.



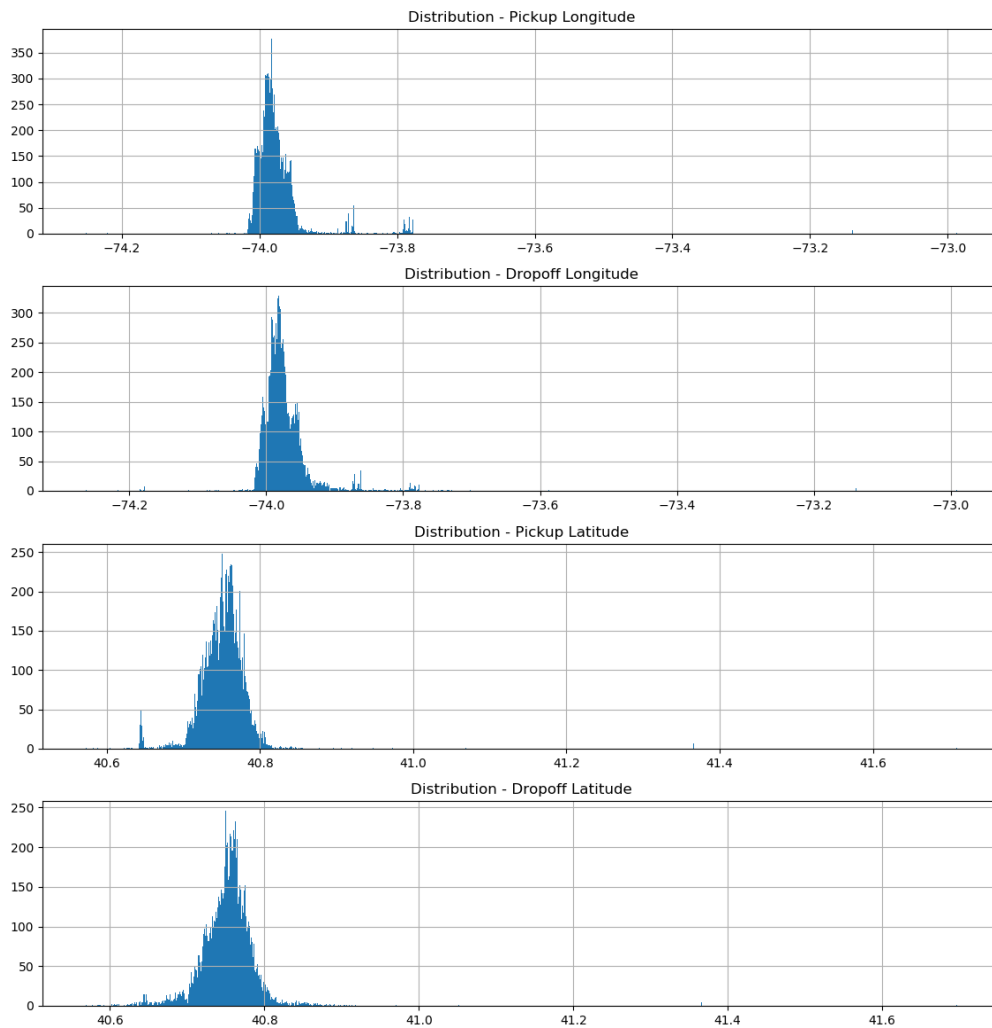
The distribution of pickup and dropoff longitude and latitude are plotted, respectively. The peak in each normal distribution indicates most of the trips are in the centre area of NYC (within or near Manhattan). The other small peaks are in accord with the coordinates of the three airports near NYC:

```
jfk_airport = (-73.786578, 40.64879)
ewr_airport = (-74.178894, 40.6928)
lgr_airport = (-73.872825, 40.77387)
```

## 2.4.2 Test Set



The distribution of passenger count of test set is similar to that of training set. The major difference is that there is no passenger count other than 1, 2, 3, 4, 5 and 6. From this, we may consider cleansing all the instances in training set with passenger count other than these numbers, to filter the noise. But, experiments have to be done before making conclusion.



The distribution of pickup and dropoff longitude and latitude of test set are also plotted, which is similar to that of training set, in terms of range and distribution pattern.

## 2.5 Location (printing on real map)

*The idea and part of the python code was sourced and developed from Kernel posted by Albert van Breemen.*

(<https://www.kaggle.com/breemen/nyc-taxi-fare-data-exploration>)

It is a good idea to plot the longitude-latitude coordinates on real maps of New York to find out more patterns in the data.

The 2 maps of NYC (nyc\_-74.5\_-72.8\_40.5\_41.8 & nyc\_-74.3\_-73.7\_40.5\_40.9) were downloaded from the links Albert provided in his kernel and were saved as .png format in local hard disk. They can be read by `plt.imread()` method. The 2 tuples BB and BB\_zoom are used to define the data and plotting range.

```
nyc_map = plt.imread('nyc_-74.5_-72.8_40.5_41.8.png')
nyc_map_zoom = plt.imread('nyc_-74.3_-73.7_40.5_40.9.png')
```

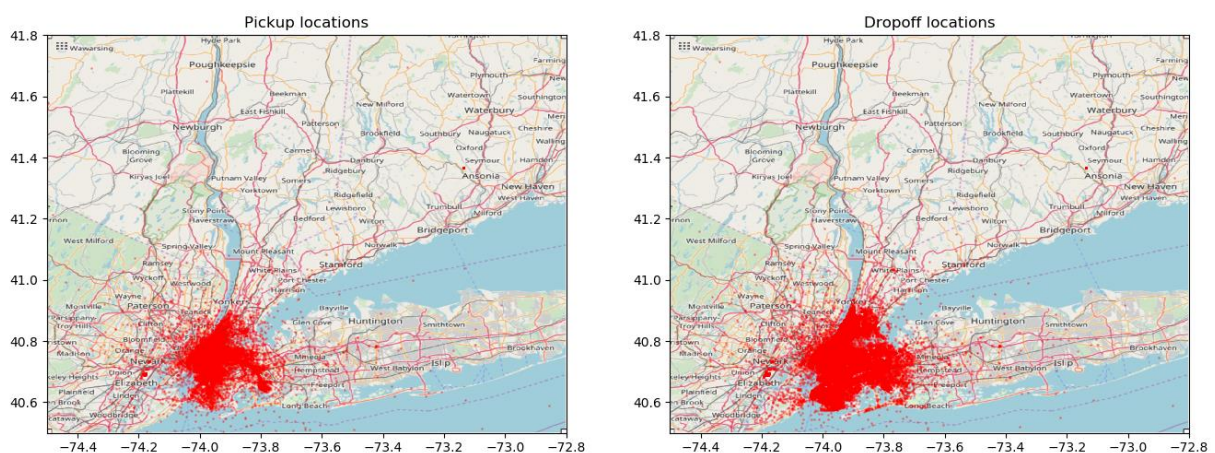
```
BB = (-74.5, -72.8, 40.5, 41.8)
BB_zoom = (-74.3, -73.7, 40.5, 40.9)
```

A function was defined to plot the data points on the maps:

```
def plot_on_map(df, BB, nyc_map, s = 10, alpha = 0.2):
    fig, axs = plt.subplots(1, 2, figsize=(16,10))
    axs[0].scatter(df.pickup_longitude, df.pickup_latitude, zorder = 1, alpha =
alpha, c = 'r', s = s)
    axs[0].set_xlim((BB[0], BB[1]))
    axs[0].set_ylim((BB[2], BB[3]))
    axs[0].set_title('Pickup locations')
    axs[0].imshow(nyc_map, zorder = 0, extent = BB)

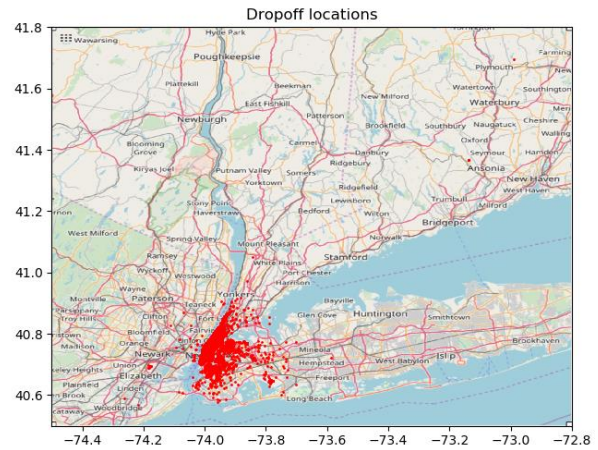
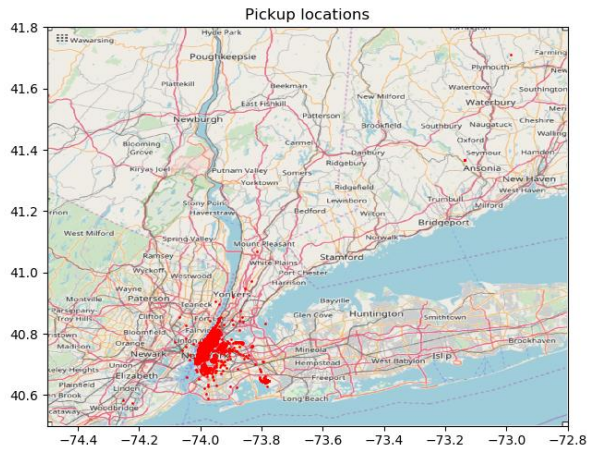
    axs[1].scatter(df.dropoff_longitude, df.dropoff_latitude, zorder = 1, alpha =
alpha, c = 'r', s = s)
    axs[1].set_xlim((BB[0], BB[1]))
    axs[1].set_ylim((BB[2], BB[3]))
    axs[1].set_title('Dropoff locations')
    axs[1].imshow(nyc_map, zorder = 0, extent = BB)
    plt.show()
```

The first 1 million rows of data in training set are plotted in the maps below, the pickup points were showed in the image on the left; the dropoff points were showed on the right.



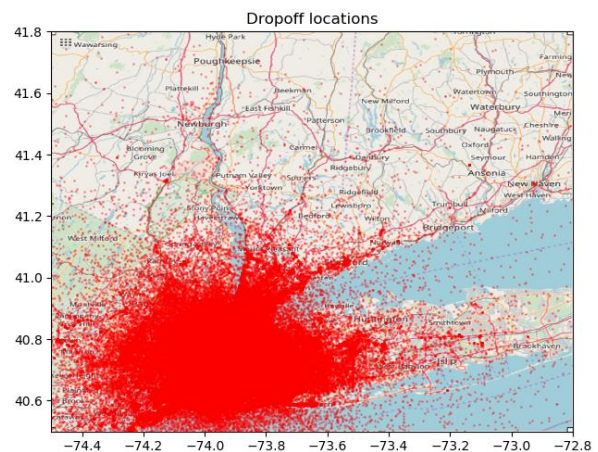
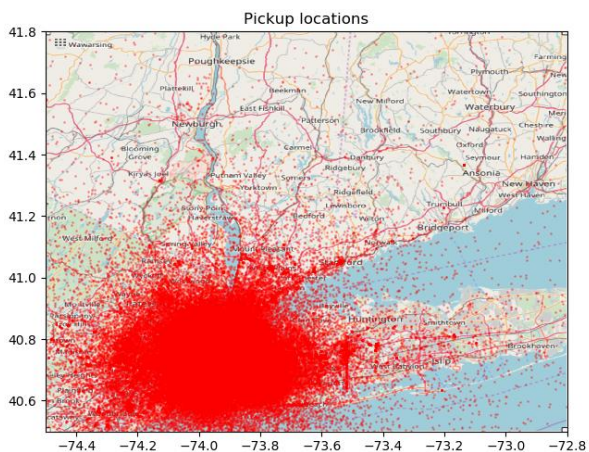
The test set was also plotted on the same map. We found most of the pickup and dropoff locations in test set are within Manhattan area or around the 3 airports.



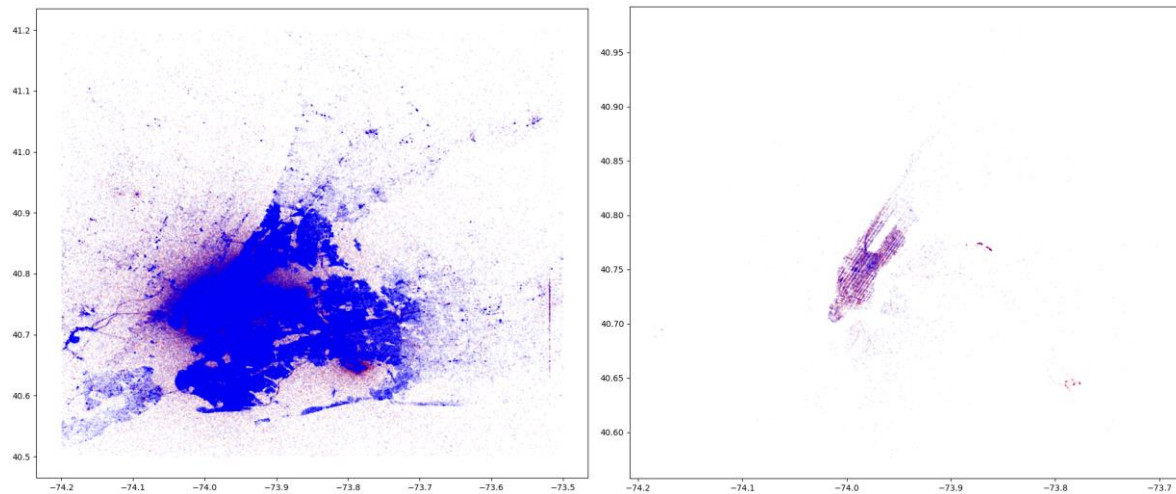


## 2.6 Data points in the water area

If plot all the 55 million trips data on the map, even though the huge number of data points make the visualization not clearer than previous ones, but we can find there are quite a lot data points in the water area, which may need to be cleansed.



If reducing the point size value and differentiate the colour by pickup and dropoff when plotting, we can get a even clearer understanding of location distribution in the raw data. The image on the left shows the training set, the one on the right shows the test set (which confirms again, most of the locations in test set are within Manhattan area).



## 2.7 Other patterns

There can be more patterns found if consider feature interactions. But a data cleansing is needed before that. So, more patterns will be discussed in part 2 of this report.

## Part 2: Developing and testing machine learning system

In this part, data cleansing, pre-processing, feature engineering and modelling process were designed and optimised to predict the fare amount in the testset.csv. In the beginning, KNN regression was selected and the performance of the initial design on leaderboard was only around 4 (top 60%). After a lot of work in feature engineering, the score was improved to 3.3 (top 35%). However, the score was no longer improved after that, even attempted for a lot of times. Finally, a new algorithm (Light Gradient Boosting Machine) was chosen and it has been proved outperform the KNN regression algorithm. The final score on the leader board is 2.99 (top 17%).

The pipeline was built using Python. Each of the single step was packaged into a function with parameters setting by arguments. The code is shown below:

```
training_data_reading(1000000)      # Read the first 1 million rows
#training_data_reading(0)           # Read all 55 million rows
test_data_reading()
data_visualization()
data_cleansing(1, 1, 0, 1, 1, 1)
feature_manipulation_trainingset()
feature_manipulation_testset()
feature_manipulation_for_both_sets()
modelling_and_submit()
```

### 3. Data Cleansing

#### 3.1 Cleansing Missing Data

When preview the raw dataset, missing data were found. So, the data was cleaned by dropping all rows with missing data first.

```
df_train = df_train.dropna(how = 'any', axis = 'rows')
```

#### 3.2 Cleansing Data by Fare Amount

There are values found in fare\_amount column which are negative. There are also positive values which are found less than the initial charge \$2.5 for taxi in New York.

(source: [http://www.nyc.gov/html/tlc/html/passenger/taxicab\\_rate.shtml](http://www.nyc.gov/html/tlc/html/passenger/taxicab_rate.shtml))

##### 3.2.1 Cleansing Data with Fare Amount < \$2.5

If we choose to remove all the data with fare\_amount less than 2.5 (including all negative value), the dataset can be filtered by:

```
df_train = df_train[df_train.fare_amount >= 2.5]
```

##### 3.2.2 Cleansing Data with Absolute Value of Fare Amount < \$2.5

If consider all the data with negative fare\_amount values got the negative sign accidentally, only the data between -2.5 and 2.5 need to be removed. All the fare\_amount data will be taken into account as long as their absolute values is larger than 2.5.

```
df_train['fare_amount'] = df_train.fare_amount.apply(lambda x: abs(x))
df_train = df_train[df_train.fare_amount >= 2.5]
```

There was 0.05055 improvement in Kaggle score if using 3.2.2 strategy, and 0.06156 improvement after using 3.2.1 strategy, comparing to not using any one of them. So, cleansing data with fare amount less than \$2.5 directly was selected.

#### 3.3 Cleansing Data by the Count of Fare Amount

From the NY Taxi Commission website, the taxi cost increases by 50 cents per 1/5 mile or 50 cents per 60 seconds after initial charge. There are also a 50-cent MTA State Surcharge, a 30-cent Improvement Surcharge, a daily 50-cent surcharge from 8pm to 6am, a \$1 surcharge from 4pm to 8pm on weekdays. So, from the initial thinking, the correct fare values should be discrete number starting from 2.5, and with only one decimal. All the other fare amount numbers (especially the ones with more than one decimal) may somehow be incorrect. However, some of the values with more than one decimal are also popular, like \$57.33, there are 239,985 times of it in the dataset. This may mean other popular fare amounts with more than 1 decimal (only the top 5 are listed below) should also not be ignored. After searching online, people pay tips when taking taxi in NYC, so the fare can be any number.

Fare Amount	Count
57.33	239,985
49.57	107,512
57.54	34,564

49.15	32,061
35.33	16,467

However, we can try to remove the rows with fare\_amount appears for only 1 time in the more than 55 million cases, the code is as below:

```
df_count = pd.DataFrame(df_train.groupby('fare_amount').key.count())
df_count.columns.values[0] = 'fare_amount_count'
df_train = pd.merge(df_train, df_count, how = 'left', on = 'fare_amount')
del df_count
df_train = df_train[df_train.fare_amount_count > 1]
df_train.drop('fare_amount_count', axis = 1, inplace = True)
```

There was 0.04706 improvement in Kaggle score if cleanse data with the count of fare amount is only 1, comparing to not cleansing it. But if remove the data with fare amount count larger than 1 (like 2, 3, 4...), the Kaggle performance become worse than choosing 1.

### 3.4 Cleansing Data by Longitude & Latitude

After checking the 9914 rows of data in test.csv, some of the training data point are too far out of the longitude and latitude range in test set. Those data can be removed, because they may not be referred to when predict the fare amount in test set.

So, the maximum and minimum were set as below:

```
longitude_min    = -74.3
longitude_max     = -72.9
latitude_min     = 40.5
latitude_max     = 41.8
```

The code to filter the training set:

```
df_train = df_train[df_train.pickup_longitude > longitude_min]
df_train = df_train[df_train.pickup_longitude < longitude_max]
df_train = df_train[df_train.dropoff_longitude > longitude_min]
df_train = df_train[df_train.dropoff_longitude < longitude_max]
df_train = df_train[df_train.pickup_latitude > latitude_min]
df_train = df_train[df_train.pickup_latitude < latitude_max]
df_train = df_train[df_train.dropoff_latitude > latitude_min]
df_train = df_train[df_train.dropoff_latitude < latitude_max]
```

### 3.5 Cleansing Trips with Same Pick-up & Drop-off Location

There are also some points were found with exactly the same pick-up location and drop-off location, which means the GPS may not work property. Or, they are return trips. But it is hard to say where did they go before coming back just from the data provided. So, these data may affect the accuracy of the model if use.

```
df_train = df_train[~((df_train.pickup_longitude == df_train.dropoff_longitude)
& (df_train.pickup_latitude == df_train.dropoff_latitude))]
```

### 3.6 Cleansing Data with Pick-up or Drop-off Location in Water Area

The function below was learned from Kernel posted by Albert van Breemen, which is a good idea to remove trips with pick-up and drop-off location in the water area using image processing technique.

(Source: <https://www.kaggle.com/breemen/nyc-taxi-fare-data-exploration>)

```
def remove_datapoints_from_water():
    global df_train

    def lonlat_to_xy(longitude, latitude, dx, dy, BB):
        return (dx * (longitude - BB[0]) / (BB[1] - BB[0])).astype('int'),
            (dy - dy * (latitude - BB[2]) / (BB[3] - BB[2])).astype('int')

    BB = (-74.5, -72.8, 40.5, 41.8)
    nyc_mask = plt.imread('https://aiblog.nl/download/
        nyc_mask-74.5_-72.8_40.5_41.8.png')[:, :, 0] > 0.9
    pickup_x, pickup_y = lonlat_to_xy(df_train.pickup_longitude,
        df_train.pickup_latitude, nyc_mask.shape[1], nyc_mask.shape[0], BB)
    dropoff_x, dropoff_y = lonlat_to_xy(df_train.dropoff_longitude,
        df_train.dropoff_latitude, nyc_mask.shape[1], nyc_mask.shape[0], BB)
    idx = nyc_mask[pickup_y, pickup_x] & nyc_mask[dropoff_y, dropoff_x]
    return df_train[idx]
df_train = remove_datapoints_from_water()
```

There was 0.04532 improvement in Kaggle score if cleansing data with either pickup or dropoff location in water area comparing to not cleansing them.

### 3.7 Cleansing Data with passenger count out of range of test set

There are data rows in training set with passenger count less than 1 and larger than 6. Especially, there are some cases with passenger count as 0. This is probably because the driver forgot to input the passenger count, which may bring noise into the model. And for a count number larger than 6, it seems impossible by common sense.

```
df_train = df_train[df_train.passenger_count <= 6]
df_train = df_train[df_train.passenger_count >= 1]
```

### 3.8 Result

After data cleansing, 53,464,464 rows of data were left, around 1.959 million trips were removed.

19:50:50	Data Cleansing 1	After drop rows with missing data :	55423480
19:50:55	Data Cleansing 2	After drop fare_amount < 2.5 directly:	55418733
19:52:11	Data Cleansing 3	After drop fare_amount count <= 1 :	55415383
19:52:43	Data Cleansing 4	After drop data out of test_set coordinate range :	54238182
19:52:48	Data Cleansing 5	After drop pickup dropoff same coordinate :	53665262
19:52:58	Data Cleansing 6	After drop data in water :	53654855
19:53:05	Data Cleansing 7	After drop passenger count <1 and >6:	53464464



## 4. Feature Manipulation

In the raw data, we have 7 effective raw features. They can be separated into 4 groups:

- a) Time-related feature: pickup\_datetime
- b) Space-related features: pickup\_longitude ,  
pickup\_latitude,  
dropoff\_longitude,  
dropoff\_latitude
- c) Counting feature: passenger\_count
- d) Feature to be predicted: fare\_amount

9 more time-related features and 34 more space-related features are developed and appended to the DataFrame from the original 1 time-related and 4 space-related features, respectively.

### 4.1 Time-related features manipulation

If come back to the business understanding of the project, taxi fare is strongly related to when you take the taxi. Is it a Monday morning, a Friday afternoon or the evening before a new year? Or just a normal weekday around lunch time? The traffic affects the taxi fare pretty much. Also, for some special time, special day, the taxi rates are different.

From the NYC government website, there are some rules and regulations related to time and date:

- i. The initial charge is \$2.50 since May 2004  
All the pickup\_datetime in training set and test set are later than May 2004, so \$2.5 applies to all the cases in this study.  
(Source: <http://www.schallerconsult.com/taxi/taxifb.pdf>)
- ii. Plus 50 cents per 60 seconds in slow traffic or when the vehicle is stopped.  
Traffic jam pattern should be explored, and traffic jam period should be taken into account.
- iii. There is a daily 50-cent surcharge from 8 p.m. to 6 a.m.  
The 24 hours in a day should be separated into several periods, one of them is 8 p.m. to 6 a.m.
- iv. There is a \$1 surcharge from 4pm to 8pm on weekdays, excluding holidays.  
4pm to 8 pm should be another period of a day. And weekdays, weekends and holidays should also be taken into account.
- v. There is a \$4.50 rush hour surcharge to/from JFK and any location in Manhattan from 4 PM to 8 PM weekdays, excluding legal holidays.
- vi. There are Group Ride pickup times from 6am to 10am from Monday to Friday (excluding holidays).

`add_time_features()` function was defined in Python to add 9 more features related to time. They are:

- 1) hour\_of\_day
- 2) day\_of\_week
- 3) day\_of\_month
- 4) day\_of\_year
- 5) week\_of\_year
- 6) month
- 7) quarter
- 8) year
- 9) period\_of\_day

```

def add_time_features():
    df_train['pickup_datetime'] = df_train['pickup_datetime'].replace(' UTC', '')
    df_train['pickup_datetime'] = pd.to_datetime(df_train['pickup_datetime'],
format='%Y-%m-%d %H:%M:%S')
    df_train['hour_of_day']      = df_train.pickup_datetime.dt.hour
    df_train['day_of_week']      = df_train.pickup_datetime.dt.weekday
    df_train['day_of_month']     = df_train.pickup_datetime.dt.day
    df_train['day_of_year']      = df_train.pickup_datetime.dt.dayofyear
    df_train['week_of_year']     = df_train.pickup_datetime.dt.weekofyear
    df_train['month']            = df_train.pickup_datetime.dt.month
    df_train['quarter']          = df_train.pickup_datetime.dt.quarter
    df_train['year']             = df_train.pickup_datetime.dt.year
    def period_of_day(hour):
        if hour >= 6 and hour < 16:
            return 1
        elif hour >= 16 and hour < 20:
            return 2
        elif hour >= 20 or hour < 6:
            return 3
    df_train['period_of_day']     = df_train.pickup_datetime.apply(lambda t:
period_of_day(t.hour))

```

The 3 period options in `period_of_day` are 4 p.m. to 8 p.m., 8 p.m. to 6 a.m. and 6 a.m. to 4 p.m., which are related to the surcharge policies mentioned in the NYC taxi fare rules above.

## 4.2 Space-related features manipulation

As a common sense, the taxi fares are related to the distance the passengers travel. So, the space-related features are also key to the model. However, in the raw data, the only fact we know related to space are the starting location and the ending location. There is no information about how the taxi go between the 2 points. It may go directly from point A to point B in a straight line, or it has to go through a bridge 10km away even if the direct distance between A and B is only 2km but on the 2 sides of Hudson River. So, the areas (suburbs, districts, cities or even stats) the pickup and dropoff points located in, the locations of bridges which connected two areas may also need to be taken into account. Of course, before that the NYC should be separated into zones first.

From the NYC government website, there are also some rules and regulations related to some special locations and routes:

- i. Plus 50 cents per 1/5 mile after the initial charge.
- ii. There is a 50-cent MTA State Surcharge for all trips that end in New York City or Nassau, Suffolk, Westchester, Rockland, Dutchess, Orange or Putnam Counties.
- iii. Passengers must pay all bridge and tunnel tolls.
- iv. It is a metered fare to & from LaGuardia Airport
- v. There is a flat fare of \$52 plus tolls, a 50-cent MTA State Surcharge, a 30-cent Improvement Surcharge to/from JFK and any location in Manhattan
- vi. To Newark Airport is a metered fare plus a \$17.50 Newark Surcharge and a 30-cent Improvement Surcharge. Passenger is responsible for paying all roundtrip tolls.
- vii. The metered fare is double the amount from the City limit to your destination.
- viii. Passenger is responsible for paying all roundtrip tolls.

`add_distance_features()` function was defined in Python to add 34 more features related to the location, distance and direction between the pickup and dropoff points. They can be separated in to several categories:



### 4.2.1 Longitude/Latitude Distance in Straight Line

```
df_train['longitude_distance'] = abs(df_train['pickup_longitude'] -
df_train['dropoff_longitude'])
df_train['latitude_distance'] = abs(df_train['pickup_latitude'] -
df_train['dropoff_latitude'])
df_train['itude_dist'] = (df_train['longitude_distance'] ** 2 +
df_train['latitude_distance'] ** 2) ** .5
```

To standardize the distance, several features can be developed from 'itude\_dist':

```
df_train['itude_dist_sin'] = np.sin((df_train['longitude_distance'] ** 2
+ df_train['latitude_distance'] ** 2) ** .5) # sin(itude_dist)
df_train['itude_dist_cos'] = np.cos((df_train['longitude_distance'] ** 2
+ df_train['latitude_distance'] ** 2) ** .5) # cos(itude_dist)
df_train['itude_dist_sin_sqrd'] = np.sin((df_train['longitude_distance'] ** 2
+ df_train['latitude_distance'] ** 2) ** .5) ** 2 # sin²(itude_dist)
df_train['itude_dist_cos_sqrd'] = np.cos((df_train['longitude_distance'] ** 2
+ df_train['latitude_distance'] ** 2) ** .5) ** 2 # cos²(itude_dist)
```

### 4.2.2 Haversine Distance and Bearing Angle

Earth is a sphere, so haversine distance and bearing angle may be more accurate if use for modelling. The formula and code were developed from the source <https://www.movable-type.co.uk/scripts/latlong.html>

```
R = 6371e3 # Equatorial radius (m)
phi1 = np.radians(df_train['pickup_latitude'])
phi2 = np.radians(df_train['dropoff_latitude'])
delta_phi = np.radians(df_train['dropoff_latitude'] - df_train['pickup_latitude'])
delta_lmd = np.radians(df_train['dropoff_longitude'] - df_train['pickup_longitude'])
a = np.sin(delta_phi / 2) * np.sin(delta_phi / 2)
+ np.cos(phi1) * np.cos(phi2)
* np.sin(delta_lmd / 2) * np.sin(delta_lmd / 2)
c = 2 * np.arctan2(a ** 0.5, (1-a) ** 0.5)
d = R * c
df_train['haversine'] = d # Haversine distance (m)

y = np.sin(delta_lmd * np.cos(phi2))
x = np.cos(phi1) * np.sin(phi2)
- np.sin(phi1) * np.cos(phi2) * np.cos(delta_lmd)
df_train['bearing'] = np.degrees(np.arctan2(y, x)) # Bearing angle
```

### 4.2.3 Separating the NYC into Boxes (Blocks)

```
def longitude_to_box(longitude, long_resolution, lat_resolution):
    long_idx = math.floor((longitude - longitude_min)
/ (longitude_range / long_resolution))
    return long_idx

def latitude_to_box(latitude, long_resolution, lat_resolution):
    lat_idx = math.floor((latitude - latitude_min)
/ (latitude_range / lat_resolution))
    return lat_idx

df_train['pickup_long_blk_idx'] = df_train.apply(lambda row:
longitude_to_box(row['pickup_longitude'], 600, 600), axis = 1)
df_train['pickup_lat_blk_idx'] = df_train.apply(lambda row:
latitude_to_box(row['pickup_latitude'], 600, 600), axis = 1)
df_train['dropoff_long_blk_idx'] = df_train.apply(lambda row:
longitude_to_box(row['dropoff_longitude'], 600, 600), axis = 1)
df_train['dropoff_lat_blk_idx'] = df_train.apply(lambda row:
```

```
latitude_to_box(row['dropoff_latitude'], 600, 600), axis = 1)
```

The idea of separating NYC into zones(blocks) come from the model selected initially: KNN (K-Nearest Neighbours). Since from the common sense, there is high possibility that similar trips (similar time, similar pickup and dropoff location) in test set has been done by other people previously or later on in the training set. Also, there are huge number of instances in training data set to refer to, the k-nearest neighbours here can be considered as k-most similar trips.

Time can be digitized by varies of time units. Location here is also needed to be discretized. We can imagine that there should not be much difference in price for different people get on a taxi in a 50m\*50m zone A in Manhattan and head to a 50m\*50m zone B in Brooklyn at a similar time. The model just needs to find out the nearest trips and then can predict the price.

So, here, the NYC is separated into  $600*600=360000$  blocks (100\*100 to 2000\*2000 were tested by increasing 100 for each time, 600\*600 gave the best result). 2 functions were defined to calculate the boxes number from longitude and latitude. Initially, there were only 2 new parameters added pickup\_blk\_idx (from 1 to 360000) and dropoff\_blk\_idx (from 1 to 360000). However, if consider the effect of the continuity of the block number on the accuracy of the model, 4 parameters have to be used here to make sure the model can find the nearest trips efficiently.

#### 4.2.4 Popular areas in NYC

There are several hot areas in NYC which may help to predict the taxi fare, which has been confirmed by the taxi fare rules mentioned above. So, NYC is separated into 5 areas: Manhattan, JFK Airport, LaGuardia Airport, Newark Airport and other area. There are several special taxi fare rules related to these locations.

First of all, a within\_manhattan(x, y) function was defined, which gives the answer whether the longitude and latitude input is in the Manhattan area or not.

```
def within_manhattan(x, y):
    if ((y < x * 1.929503 + 183.553291) & \
        (y > x * 0.215615 + 56.657966) & \
        (y > x * 8.238748 + 650.131497) & \
        (y > x * 1.387465 + 143.369265) & \
        (y < x * (-5.700678) - 380.638322)) == True:
        return True
    else:
        return False
```

Then, the logic to judge whether a location can be defined as in the 3 airports areas were also defined:

```
def itude_dist(x1, y1, x2, y2):
    return ((x1 - x2) ** 2 + (y1 - y2) ** 2) ** 0.5

if within_manhattan(x, y) == True:
    return 0
elif itude_dist(x, y, jfk[0], jfk[1]) < 0.0437:
    return 1
elif itude_dist(x, y, ewr[0], ewr[1]) < 0.0421:
    return 2
elif itude_dist(x, y, lgr[0], lgr[1]) < 0.0187:
    return 3
else:
    return 4
```

The longitude and latitude of the 3 airports are:

```
jfk = (-73.786578, 40.64879)
ewr = (-74.178894, 40.6928)
lgr = (-73.872825, 40.77387)
```

The radius(from each airport) numbers are measured and finalised in Google Earth Pro.

Then, 2 new features are added:

```
df_train['pickup_popular_zone' ] = df_train.apply(lambda row:
popular_zones(row['pickup_longitude' ], row['pickup_latitude' ]), axis = 1)
df_train['dropoff_popular_zone'] = df_train.apply(lambda row:
popular_zones(row['dropoff_longitude'], row['dropoff_latitude']), axis = 1)
```

The value of these two features means:

Pickup/Dropoff Zone	Area
0	Manhattan
1	JFK Airport
2	LaGuardia Airport
3	Newark Airport
4	other area

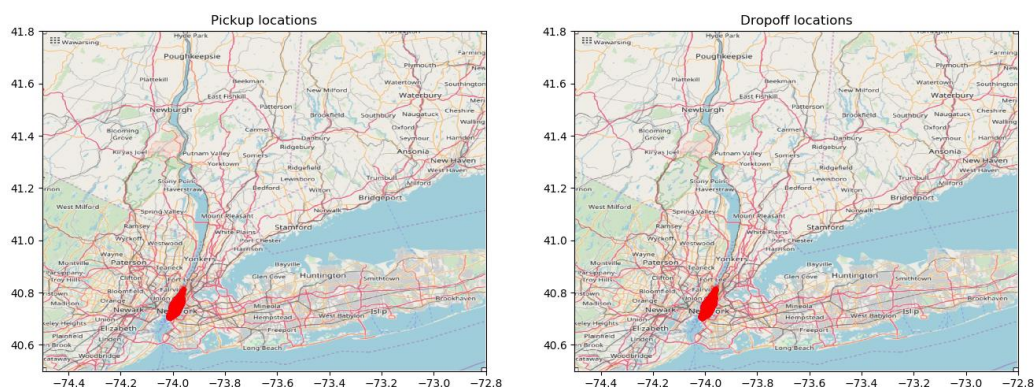
Because there are 5 pickup popular zones and 5 dropoff popular zones, there are 25 different kinds of trips between each 2 of them. So, another new feature is defined to combine the two features above:

```
def trip_route(pickup_zone, dropoff_zone):
    return pickup_zone * 5 + dropoff_zone

df_train['trip_route' ] = df_train.apply(lambda row:
trip_route(row['pickup_popular_zone'], row['dropoff_popular_zone']), axis = 1)
```

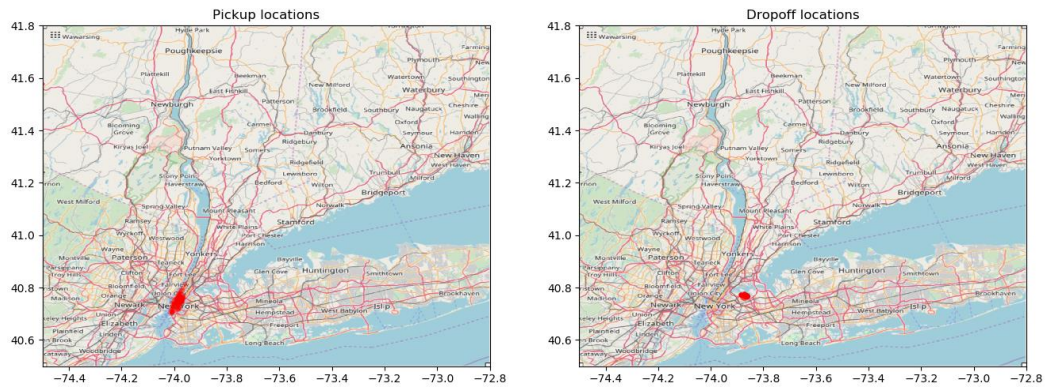
If plot the trips within Manhattan, we can see the Manhattan area in both the pickup locations and dropoff locations map are highlighted.

(pickup\_popular\_zone = 0, dropoff\_popular\_zone = 0, trip\_route = 0)

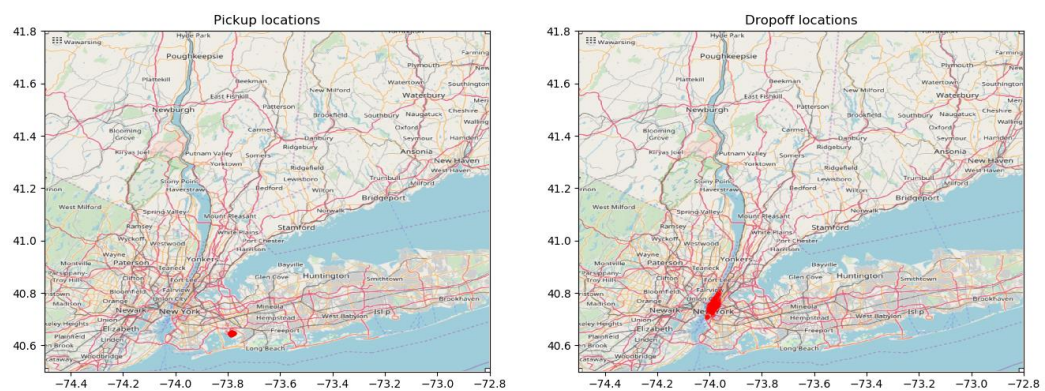


If plot the trips from Manhattan to LaGuardia Airport:

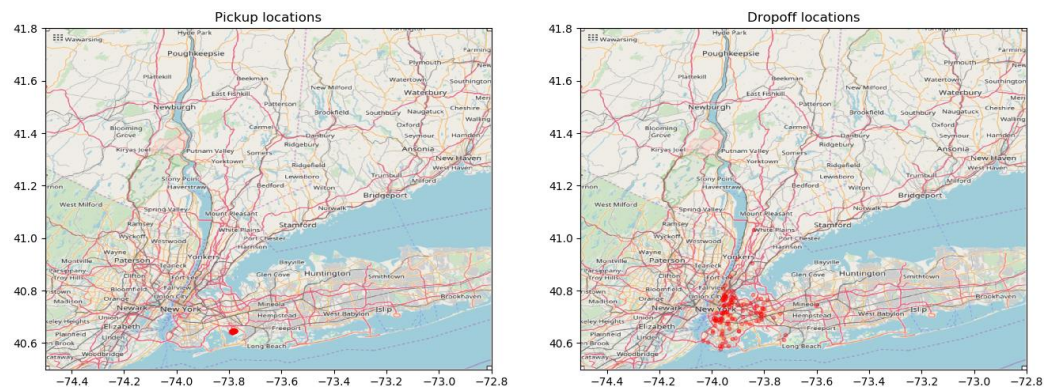
(pickup\_popular\_zone = 0, dropoff\_popular\_zone = 2, trip\_route = 2)



If plot the trips from JFK Airport to Manhattan:  
(pickup\_popular\_zone = 1, dropoff\_popular\_zone = 0, trip\_route = 5)



If plot the trips from JFK Airport to other areas in NYC:  
(pickup\_popular\_zone = 1, dropoff\_popular\_zone = 4, trip\_route = 9)



#### 4.2.5 Manhattan Distance

Manhattan distance needs to be induced to help the model predict more accurately, especially in this car driving route related case.

From the definition, it is the distance between two points measured along axes at right angles. So, several new features can be defined and added:

```
df_train['itude_mht_distance'] = df_train.apply(lambda row:
(np.abs(row['dropoff_latitude'] - row['pickup_latitude'])) +
```



```

np.abs(row['dropoff_longitude'] - row['pickup_longitude'])), axis = 1)

df_train['mht_dist_pkp_to_ctr'] = mht_dist(nyc[0], nyc[1],
      df_train['pickup_longitude'], df_train['pickup_latitude'])
df_train['mht_dist_ctr_to_dpj'] = mht_dist(nyc[0], nyc[1],
      df_train['dropoff_longitude'], df_train['dropoff_latitude'])
df_train['mht_dist_pkp_to_jfk'] = mht_dist(jfk[0], jfk[1],
      df_train['pickup_longitude'], df_train['pickup_latitude'])
df_train['mht_dist_jfk_to_dpj'] = mht_dist(jfk[0], jfk[1],
      df_train['dropoff_longitude'], df_train['dropoff_latitude'])
df_train['mht_dist_pkp_to_ewr'] = mht_dist(ewr[0], ewr[1],
      df_train['pickup_longitude'], df_train['pickup_latitude'])
df_train['mht_dist_ewr_to_dpj'] = mht_dist(ewr[0], ewr[1],
      df_train['dropoff_longitude'], df_train['dropoff_latitude'])
df_train['mht_dist_pkp_to_lgr'] = mht_dist(lgr[0], lgr[1],
      df_train['pickup_longitude'], df_train['pickup_latitude'])
df_train['mht_dist_lgr_to_dpj'] = mht_dist(lgr[0], lgr[1],
      df_train['dropoff_longitude'], df_train['dropoff_latitude'])

```

#### 4.2.6 Manhattan 60-degree Distance

If search the Manhattan area on Google Earth, we can see almost all the streets and avenues in Manhattan and nearby area are in the direction with an angle of 60 degree and 150 degree to equator.



So, the Manhattan distance can be optimised by another orthogonal coordinate system. The direction of x'-axis is 60/240 degree; the direction of y'-axis is 150/330 degree in the original x-y coordinate system.

A new group of Manhattan distances related features were developed:

```

def itude_mht_dist_60_deg(x1, y1, x2, y2):
    if x1 != x2:
        k = (y1 - y2) / (x1 - x2)
        alfa = math.degrees(math.atan(k))
    else:
        alfa = 90
    if alfa < 0:
        alfa = 180 + alfa
    d = math.sqrt(math.pow((x1 - x2), 2) + math.pow((y1 - y2), 2))
    if alfa >= 0 and alfa < 90:
        return (d * (math.sin(math.radians(abs(alfa - 60))) +
            math.cos(math.radians(abs(alfa - 60)))))
    elif alfa >= 90 and alfa < 180:

```

```

        return (d * (math.sin(math.radians(abs(alfa -150)))) +
                math.cos(math.radians(abs(alfa - 150)))))

df_train['itude_mht_dist_60_deg'] = df_train.apply(lambda row:
        itude_mht_dist_60_deg(row['pickup_longitude'], row['pickup_latitude'],
        row['dropoff_longitude'], row['dropoff_latitude']) , axis = 1)

df_train['mht60_dist_pkp_to_ctr'] = df_train.apply(lambda row:
        itude_mht_dist_60_deg(nyc[0], nyc[1], row['pickup_longitude'],
        row['pickup_latitude']), axis = 1)
df_train['mht60_dist_ctr_to_dpj'] = df_train.apply(lambda row:
        itude_mht_dist_60_deg(nyc[0], nyc[1], row['dropoff_longitude'],
        row['dropoff_latitude']), axis = 1)
df_train['mht60_dist_pkp_to_jfk'] = df_train.apply(lambda row:
        itude_mht_dist_60_deg(jfk[0], jfk[1], row['pickup_longitude'],
        row['pickup_latitude']), axis = 1)
df_train['mht60_dist_jfk_to_dpj'] = df_train.apply(lambda row:
        itude_mht_dist_60_deg(jfk[0], jfk[1], row['dropoff_longitude'],
        row['dropoff_latitude']), axis = 1)
df_train['mht60_dist_pkp_to_ewr'] = df_train.apply(lambda row:
        itude_mht_dist_60_deg(ewr[0], ewr[1], row['pickup_longitude'],
        row['pickup_latitude']), axis = 1)
df_train['mht60_dist_ewr_to_dpj'] = df_train.apply(lambda row:
        itude_mht_dist_60_deg(ewr[0], ewr[1], row['dropoff_longitude'],
        row['dropoff_latitude']), axis = 1)
df_train['mht60_dist_pkp_to_lgr'] = df_train.apply(lambda row:
        itude_mht_dist_60_deg(lgr[0], lgr[1], row['pickup_longitude'],
        row['pickup_latitude']), axis = 1)
df_train['mht60_dist_lgr_to_dpj'] = df_train.apply(lambda row:
        itude_mht_dist_60_deg(lgr[0], lgr[1], row['dropoff_longitude'],
        row['dropoff_latitude']), axis = 1)

```

It has been proved, there was around 0.02 improvement in Kaggle score if using Manhattan 60-degree Distance comparing to using Manhattan Distance.

In the future, for a more precise modelling, the feature of Manhattan 60-degree distance can be used on to only the trips within Manhattan (both the pickup and dropoff points are within Manhattan).

## 5. Feature Aggregation

Feature aggregation is another good method to be involved into this project, because if we compare the features we developed above to the NYC taxi fare rules and regulations published online, we can find that those features do not work alone.

For example:

1. In the rule “There is a \$1 surcharge from 4pm to 8pm on weekdays, excluding holidays.”, the `hour_of_day`, `day_of_week` and `day_of_year` work together.
2. In the rule “It is a flat fare of \$52 plus tolls if to and from JFK and any location in Manhattan. There are a 50-cent MTA State Surcharge, a 30-cent Improvement Surcharge, and a \$4.50 rush hour surcharge (4 PM to 8 PM weekdays, excluding legal holidays).”, many time and space related features work together.

The function below was learned and developed from Kernel posted by Nick Brooks, which helped to perform better in Kaggle score.

(Source: <https://www.kaggle.com/breemen/nyc-taxi-fare-data-exploration>)

```
def time_agg(vars_to_agg, vars_be_agg):
    for var in vars_to_agg:
        agg = df_train.groupby(var)[vars_be_agg].agg(
            ["sum", "mean", "std", "skew", percentile(80), percentile(20)])
        if isinstance(var, list):
            agg.columns = pd.Index(["fare_by_" + "_".join(var) + "_" +
                                    str(e) for e in agg.columns.tolist()])
        else:
            agg.columns = pd.Index(["fare_by_" + var + "_" +
                                    str(e) for e in agg.columns.tolist()])
        df_train = pd.merge(df_train, agg, on = var,
                            how = "left").set_index(df_train.index)
        df_test = pd.merge(df_test, agg, on = var,
                            how = "left").set_index(df_test.index)

time_agg(vars_to_agg = ["passenger_count",
                        "day_of_week",
                        "quarter",
                        "month",
                        "year",
                        "hour_of_day",
                        ["day_of_week", "month", "year"],
                        ["hour_of_day", "day_of_week", "month", "year"]],
        vars_be_agg = "fare_amount")
```

## 6. Modelling

### 6.1 KNN (K-Nearest Neighbours) Regression

Initially, KNN regression was selected to build the model.

Training and test splits were defined with `test_size = 0.25` in `df_train` DataFrame (train.csv):

```
X = df_train[features].values
y = df_train['fare_amount'].values
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.25)
```

After that, `KNeighborsRegressor` in `sklearn.neighbors` library was selected for modelling. The `n_neighbors` parameter was finalised to 20 as it gave the best result. Also, set "distance" to `weights` resulted in a better score. `n_jobs = -1` helps to utilise all the cores in CPU to perform the modelling faster.

```
model_knn = KNeighborsRegressor(n_neighbors = 20,
                                n_jobs = -1, weights = "distance")
model_knn.fit(X_train, y_train)
```

Then, according to the description for this Kaggle competition, "the evaluation metric for this competition is the root mean-squared error". So, the average, std and rmse of the test split were calculated to check the performance of each optimization on local computer.

```
y_train_pred = model_knn.predict(X_train)
y_test_pred   = model_knn.predict(X_test)
test_avg      = np.mean(y_test - y_test_pred)
test_std      = np.std(y_test - y_test_pred)
test_rmse     = np.sqrt(mean_squared_error(y_test, y_test_pred))
```

After that, `test.csv` was read into memory, and features were calculated and appended to the `df_test` DataFrame. Then, the prediction was generated by the trained model, and saved to another `.csv` file.

```
df_test      = pd.read_csv('test.csv', parse_dates=["pickup_datetime"])
XTEST        = df_test[features].values
y_pred_final = model_knn.predict(XTEST)
submission_file = pd.DataFrame(
    {'key': df_test.key,
     'fare_amount': y_pred_final},
    columns = ['key', 'fare_amount'])
submission_file.to_csv("submission_file.csv", index = False)
```

The best score on Kaggle for this KNN algorithm was between 3.2 and 3.3, which performed not good enough. Also, because there are so many adjustable parameters in the model designed and KNN algorithm itself, which is time-consuming, another model was selected which has been proved perform better than KNN.



## 6.2 LGBM (Light Gradient Boosting Machine)

LightGBM was published by Microsoft in 2016. It is a new gradient boosting tree framework, which is highly efficient, fast, and scalable and can support many different algorithms. LightGBM is evidenced to be several times faster than existing implementations of gradient boosting trees, due to its fully greedy tree-growth method and histogram-based memory and computation optimization.

(Source: <https://www.microsoft.com/en-us/research/project/lightgbm/>)

The using of LGBM was enlightened by Nick Brooks's post on Kaggle.

First of all, LightGBM needs to load the data from DataFrame and store them in a Dataset object `training_data`. Before that `df_train` DataFrame was filtered to keep the same features as `df_test`. The Dataset object in LightGBM is very memory-efficient, due to it only need to save discrete bins.

```
y = df_train.fare_amount.copy()
df_train = df_train[df_test.columns]
training_data = lgb.Dataset(df_train, label = y, free_raw_data = False)
```

Then, the parameters of LightGBM can be setting by using a dictionary, which is defined below.

```
lgbm_params = {'task': 'train',
               'boosting_type': 'gbdt',
               'objective': 'regression',
               'metric': 'rmse'}
```

After that, the parameters related to cross-validation were set. 5-fold is used here. 10-fold was more time-consuming and during the experiments, it did not resulted in better performance if using 10-fold than using 5-fold. Shuffle was selected to be True, `random_state` was selected to be 1 in this project.

```
folds = KFold(n_splits = 5, shuffle = True, random_state = 1)
```

2 arrays of `test_split_predict` and `test_set_predict` were used to store the prediction value for the 5 times (5-fold) and calculated the average.

```
trainshape = df_train.shape
testshape = df_test.shape
test_split_predict = np.zeros(trainshape[0])
test_set_predict = np.zeros(testshape[0])
```

The next step is to train the model. `training_index` and `validation_index` are 2 lists recording the indices of data used for training and validation for each fold of training. `num_boost_round` is the number of boosting iterations. In this modelling, from experiments, since most of the time, the validation score did not improve after 4000 iterations, so the value was finalised to this number. And if the validation score did not improve for a continuous 125 iterations, the modelling will stop early. Also, by setting `verbose_eval` to 500, the model can print the metric of rmse information for every 500 iterations.

```
training_data.construct()
for training_index, validation_index in folds.split(df_train):
    LightGBM_model = lgb.train(
        params = lgbm_params,
        train_set = training_data.subset(training_index),
        valid_sets = training_data.subset(validation_index),
        num_boost_round = 4000,
```

```

        early_stopping_rounds = 125,
        verbose_eval          = 500)
test_split_predict [validation_index] = LightGBM_model.predict(
                                     training_data.data.iloc[validation_index])
test_set_predict += LightGBM_model.predict(df_test) / folds.n_splits
print("mse: ", mean_squared_error(y.iloc[validation_index], test_split_predict
[validation_index]) ** .5)

```

Finally, test\_set\_predict was transferred to DataFrame type and saved as .csv format for submission.

```

test_index= df_test.index
submission_file = pd.DataFrame(test_set_predict, columns = ["fare_amount"],
                               index = test_index)
submission_file.to_csv("LightGBM_submission_file.csv", index = True,
                       header = True)

```

## Part 3a: Team formation




The team function on Kaggle allows us data miners from all over the world to discuss and develop the best solution together. Many approaches from different members in the team can be amalgamated. Team members can share their ideas, methods, algorithms and code; also, they can share their hardware resources. For example, if any one in the team has spare GPUs resources or Cloud Machine Learning Engine for training the model, the team can share the cost together.

As a team leader in Kaggle, I can see all the submission history from any members in the team by merging with their submission history. After that, the “My Submission” page became like a list of submission history from all the members in the team. The scores for each submission can be seen and compared. I can select up to 2 submissions to be used to count towards the final leaderboard score. If 2 submissions are not selected, the system will automatically choose based on the best submission scores on the public leaderboard in the team.

The submission system also allows us to describe the idea of this submission for better communication within the team.

Ray\_Ban is the other member in my team. After more than 2 weeks struggling with the kNN algorithm, the best score I can achieve was 3.25. And there were so many parameters designed in the initial model, which made my computer run day and night. After hearing the new direction of LGBM algorithm suggested by him, I found it was worth trying it. Then I combined and modified the code from him and Nick Brooks, to make it fit in my data processing pipeline with my original data reading, visualisation, cleansing, feature manipulation part of code. And the system after combination worked well. Finally, after this team work, I got 0.26 improvement in Kaggle score, Ray\_Ban got 0.1 improvement.

Our final Kaggle score is 2.99, which ranks 174<sup>th</sup> out of 1084 contestants. (Top 17%)

 0	 0	 0
New York City Taxi Fare Pre...		174 <sup>th</sup> of 1080
5 days to go · Top 17%		

## Part 3b: Reflecting on your findings

The final machine learning model selected in this project is LightGBM due to its higher performance (lower RMSE) in prediction than other methods and algorithms tested.

LightGBM is a gradient boosting framework that uses tree-based learning algorithm. Unlike other tree-based algorithms, LightGBM grows tree vertically, leaf-wise, so that it can reduce more loss than a level-wise algorithm. Also, because of its “light”, it takes lower memory to calculate. It runs faster, especially for “big” data like the 55 million rows of raw data in this case. LightGBM worked more stably than the first algorithm used in this project - kNN. As long as the parameters are set to be the same, no matter when to run the program, the result will be the same (RMSE score on Kaggle). However, if using kNN, the results were different even if using exactly the same set of parameters. This may be because there are some random seeds or parameters in kNN algorithm itself. Finally, LightGBM gave us a high RMSE performance, which reached as low as 2.99.

Comparing to LightGBM, kNN seems easier to interpret. It is more in accord with our common sense - based on experience. When we take a taxi, there must be someone else have taken a taxi before, from a similar starting location to a similar ending location at a similar time in a similar day (weekdays, weekends, holidays or some days else but special). We just need to filter the noise and find out the k most similar trips, then we can start our prediction for a new trip. And because in this case, we have enough amount of training data to use, it was selected in the beginning as the algorithm to handle this topic. However, later on, as the model was built more and more complicated, there are more parameters emerged, and there are so many combinations among them. How to allocate weights for each parameter also affects the result pretty much and time consuming. In order to get the best result by tuning them, running speed and computer resources became the bottle neck. If more resources and time are given, this algorithm is worth exploring further.

About the feature selection in this project, the features can be separated into 3 types: time-based, space-based and count. There are more new features can be developed within each type of features. If consider the interaction between types of features, there will be even more new features to be generated. Any combination is a type of cases in real world, and how the final tree be built may affect the rules and regulations modification in the real world for different purposes on behalf of different beneficiaries. This this case, if the target is not to bias toward certain population groups, the goal will be very clear, which is to predict the fare as precise as possible (to achieve a RMSE value as low as possible).