New York City Taxi Fare Prediction

Cedric Yu[♮]

Abstract

We summarise our findings on the (expired) Kaggle competition predicting the fare amount for a taxi ride in New York City.

1 Problem Statement

In this competition, we are tasked with predicting the fare amount (inclusive of tolls) for a taxi ride in New York City with given datasets that include pickup and dropoff locations. A naive regression using just the Euclidean distance between the pickup and dropoff points would give a root-mean-squared-error (RMSE) of around $5-$8. We are asked to do better, using various techniques in feature engineering and employing more sophisticated machine learning models.

First, there are a few useful numbers. The base fare of NYC taxis is $2.50. Added to it are 50 cents per 1/5 mile when traveling above 12mph or per 60 seconds in slow traffic or when the vehicle is stopped; plus 50 cents overnight surcharge 8pm to 6am; plus $1.00 rush hour surcharge from 4pm to 8pm on weekdays, excluding holidays. Trips between JFK airport and Manhattan have a flat rate of $52 (plus a $4.50 surcharge during peak hours). The farthest driving distance within NYC is about 45 miles (between the Bronx and Staten Island), which makes the fare about $120, modulo toll fees. But then some people travel to/from New Jersey or Long Island.

1.1 Datasets

We are given the training and test datasets, which make up 5.7GB in total. The given features are:

1. `key`: the unique id associated to each instance containing the pickup datetime
2. `pickup_datetime`: timestamp value indicating when the taxi ride started
3. `pickup_longitude`: float for longitude coordinate of where the taxi ride started
4. `pickup_latitude`: float for latitude coordinate of where the taxi ride started
5. `dropoff_longitude`: float for longitude coordinate of where the taxi ride ended
6. `dropoff_latitude`: float for latitude coordinate of where the taxi ride ended
7. `passenger_count`: integer indicating the number of passengers in the taxi ride

The target is `fare_amount`: float (US) dollar amount of the cost of the taxi ride.

[♮]cedric.yu@nyu.edu. Last updated: September 16, 2021.

Preliminary Observations and Processing

The training set (`train.csv`) contains 55, 423, 855 instances, while the test set (`test.csv`) has 9, 914 instances.

The training set is a large dataset; we should avoid performing same tasks repeatedly when possible, and good memory management is crucial. To the end of the first point, we first parse the `pickup_datetime` column of both the training and test sets, extract the `year`, `month`, `day`, `weekday`, `hour` and `daylight_saving` features, and save the sets as new `csv` files. (The `daylight_saving` feature is Boolean-valued which refers to whether the instance occurred when daylight saving was in effect.) This procedure is performed in `parse_datetime.py`. To reduce memory use, every time we read the datasets into Python, we downcast the columns to either `float32`, `uint8` (`uint16` for `year`) or `bool` accordingly. The datetime-processed training set now takes up about 1.9GB of memory.

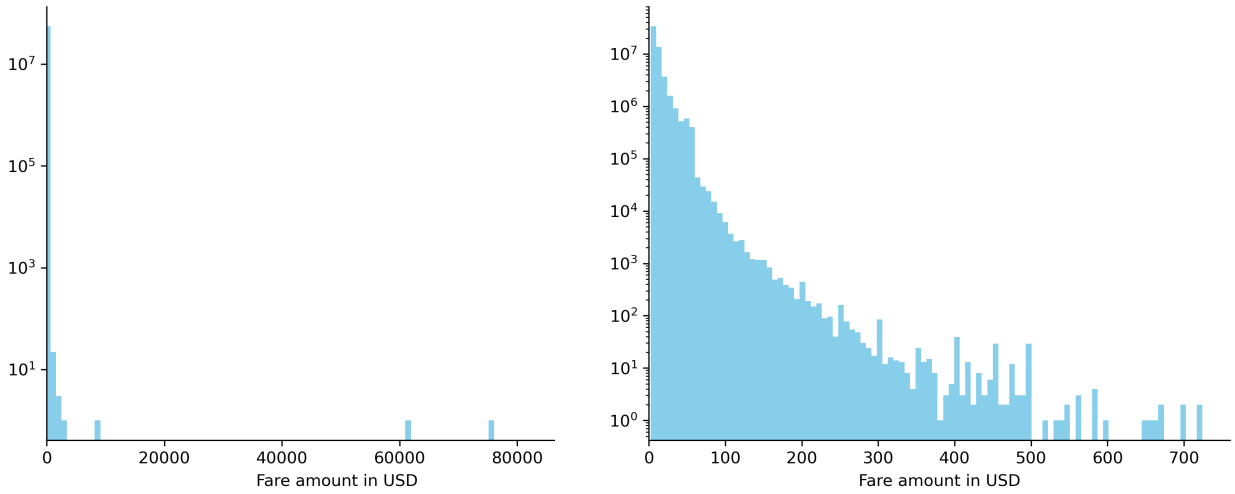
Only the training set contains `NaN`, respectively 376 instances in the `dropoff_longitude` and `dropoff_latitude` columns. We shall see that they will be gone after we discard outliers.

2 Exploratory Data Analysis

We make plots in `dataset_study.py`.

2.1 Target: `fare_amount`

From the whole training set, we first get the following histograms for the target `fare_amount`.



(a) The whole range

(b) Between $2.50 (base fare) and $800

Figure 1: Histogram of `fare_amount` from the whole training set, in log scale.

Given the real-life numbers listed in the last section, any `fare_amount` larger than $1,000 does not make sense. We make a choice to only keep instances with `fare_amount` between $2.50

(base fare) and $800 inclusive. See Figure 1b.

2.2 passenger_count

We look at the feature passenger_count. In the training set *after* keeping only instances with fare_amount between $2.50 (base fare) and $800, passenger_count has the following distribution:

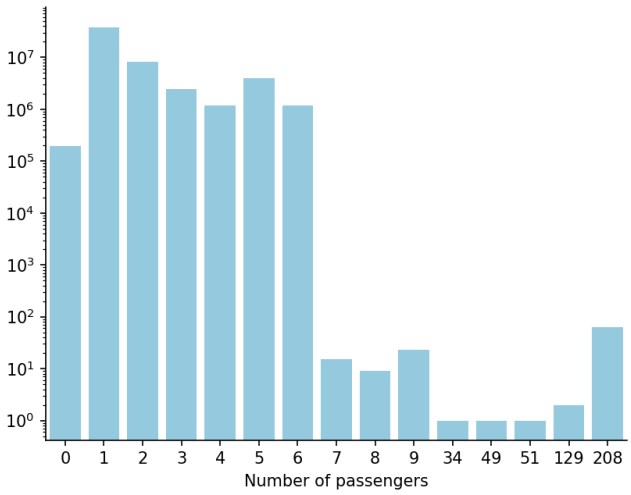


Figure 2: Histogram of passenger_count in the training set after the first truncation.

Obviously, those with passenger_count larger than ten do not make much sense. Moreover, the test set only contains passenger_count from 1 to 6. Therefore, we only keep instances with passenger_count between 1 and 6 inclusive. Then passenger_count in the training and test sets take very similar distributions; see Figure 3.

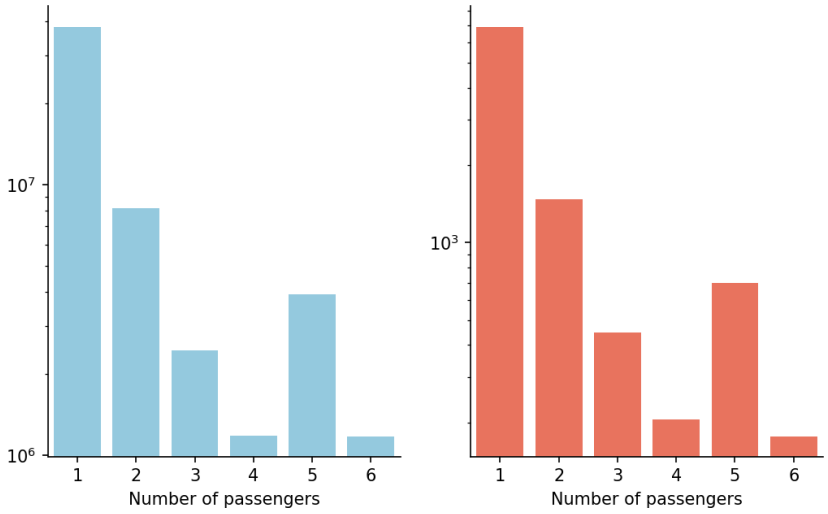
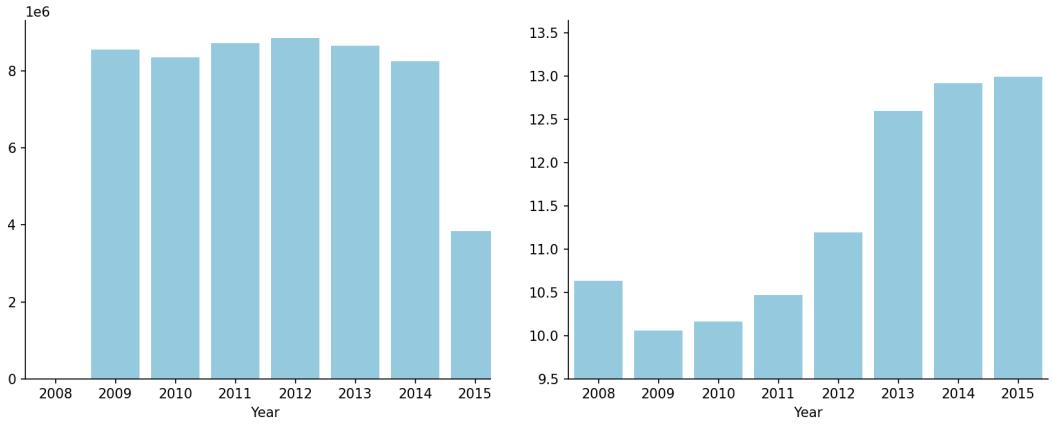


Figure 3: Histogram of `passenger_count` (between 1 and 6 inclusive) in the twice-truncated training and test sets, \$2.50≤ `fare_amount` ≤\$800.

2.3 Datetime Features

After the above two truncations (`fare_amount` and `passenger_count`), we look at the datetime features. Since the test test does not have data during year 2008, we discard those in the training set as well.

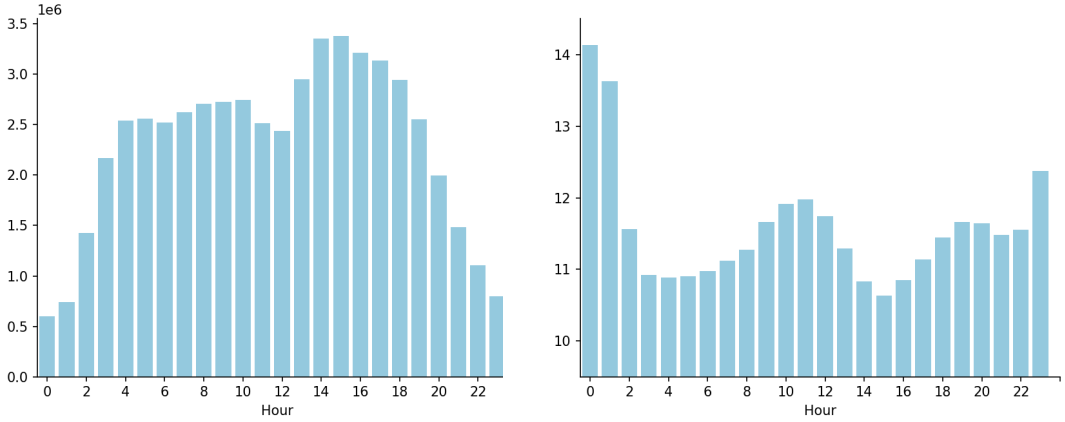
For `year`, `month`, `day` and `hour`, the respective means of `fare_amount` show more clear variations than the number counts (see e.g. Figure 4 and Figure 5). Therefore, it makes more sense to perform target (mean) encoding on these features.



(a) Ridership count by year

(b) Yearly mean `fare_amount`

Figure 4: Histograms of `year` feature from the twice-truncated training set.

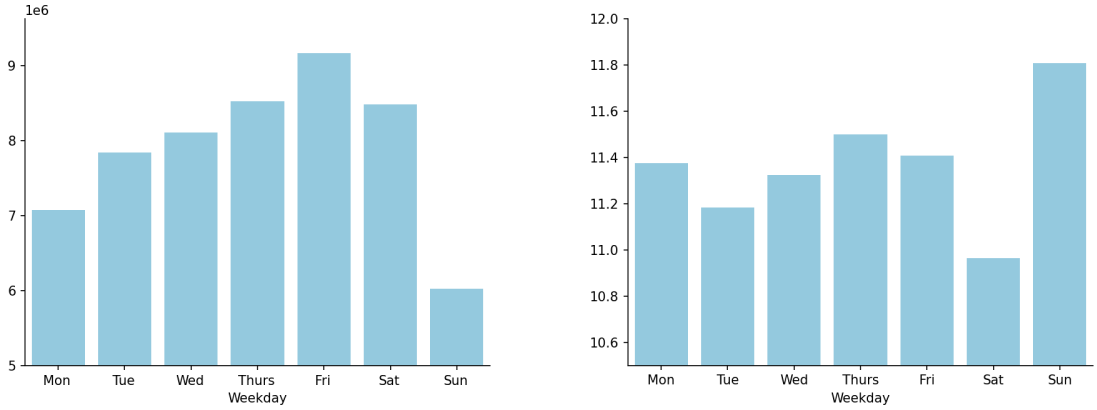


(a) Ridership count by hour

(b) Hourly mean `fare_amount`

Figure 5: Histograms of `hour` feature from the twice-truncated training set.

For `weekday`, however, the frequency plot in Figure 6 shows a much clearer trend, so we opt to perform frequency encoding on this feature.



(a) Ridership count by weekday

(b) Mean `fare_amount` by weekday

Figure 6: Histograms of `weekday` feature from the twice-truncated training set.

2.4 Geographical Data

At this point, we have performed three truncations (`fare_amount`, `passenger_count` and `year`). We now explore possible truncation based on the geographical data.

The pickup/dropoff latitudes/longitudes in the training set are as large as ± 3000, and there are some with zero values. These are obviously nonsensical outliers. However, most do fall within the approximate boundary of NYC (see Figure 7), whose coordinates are given by (40.40, -75.00) and (41.06, -73.33) that include some parts of New Jersey and Long Island; see Figure 8.

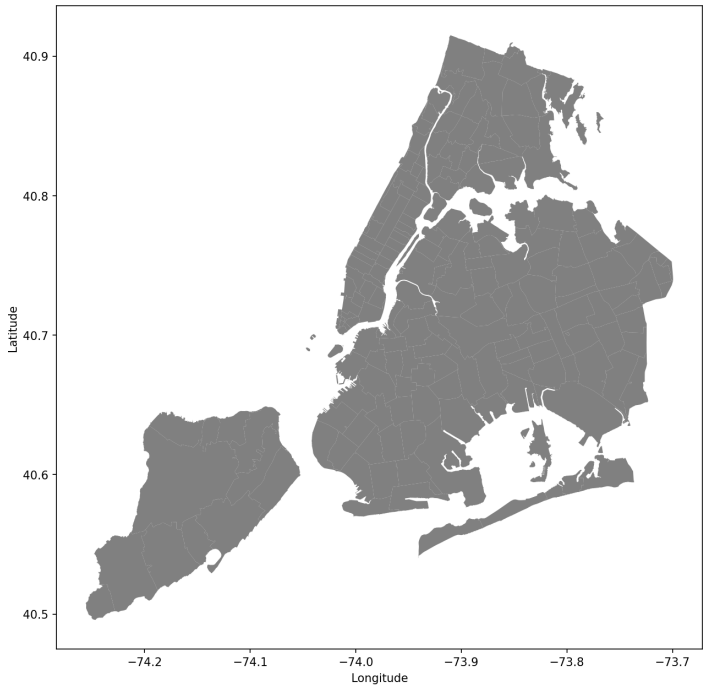
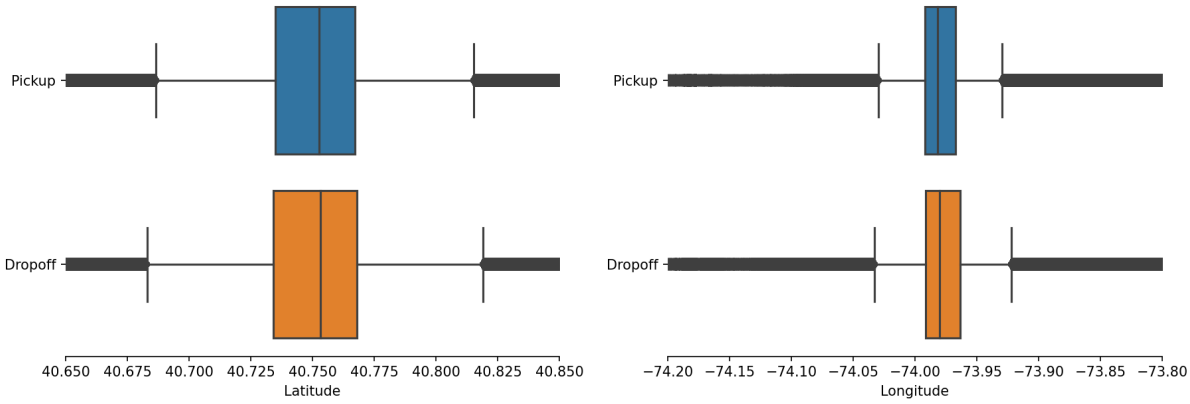


Figure 7: New York City map with zip code areas from NYC Open Data.



(a) Latitudes

(b) Longitudes

Figure 8: Box Plots of pickup/dropoff coordinates from the thrice-truncated training set.

In the next section, we will discuss how we generate new features out of the given pickup and dropoff coordinates. Among the new features obtained, here we study a) `euclidean_distance_miles`: Euclidean (flat space) geodesic distance between the end point coordinates, in miles, and b) `pickup_county` and `dropoff_county`: the county of the pickup and dropoff points.

In Figure 9, we plotted the Euclidean distance between the pickup and dropoff points in the *thrice-truncated* training set. We observe bumps at around 60 and 75 miles respectively, on an otherwise smooth plot. There is also a much narrower bump at around 10 miles.

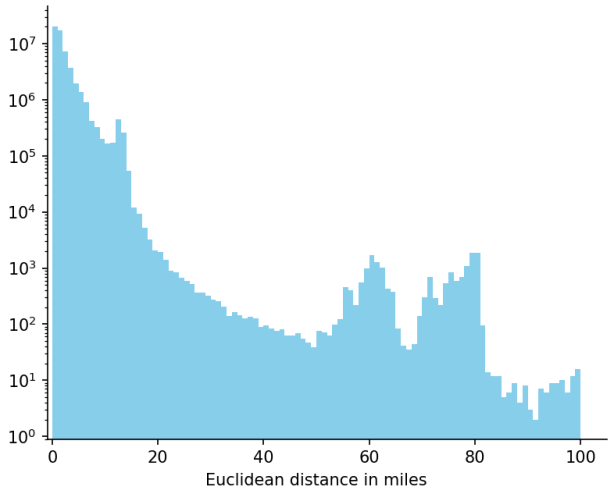
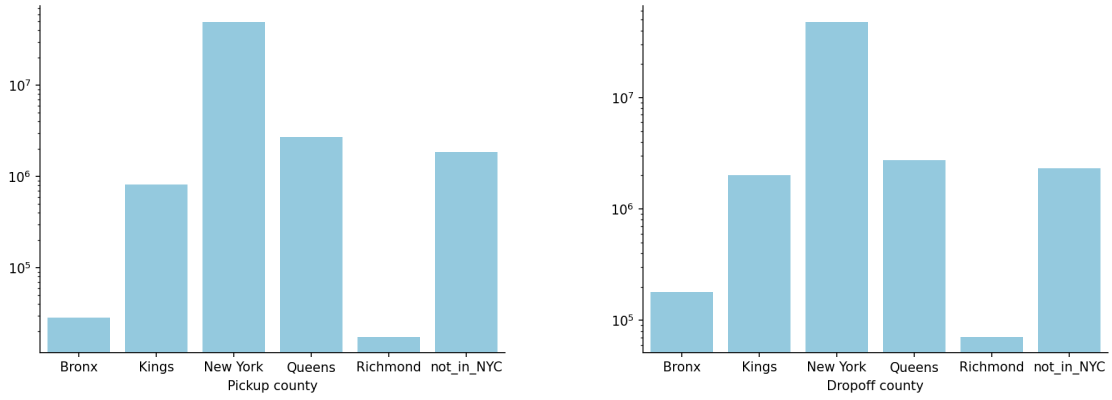


Figure 9: Euclidean distance between pickup and dropoff points in the thrice-truncated training set.

In Figure 10, we made a bar chart of the counties to which pickup/dropoff coordinates belong in the thrice-truncated training set. The (log) plots for the pickup and dropoff corrdinates look roughly the same, with New York county (Manhattan) having the highest counts.



(a) Pickup

(b) Dropoff

Figure 10: Counties to which pickup/dropoff coordinates belong in the thrice-truncated training set.

Informed by Figure 8, we make the final truncation of the training dataset: only keep instances with latitude between 40.40 and 41.06, and longitude between -75.00 and -73.33.

2.5 Discarding Outliers

In summary, we have performed four truncations on the training set, keeping only the instances with

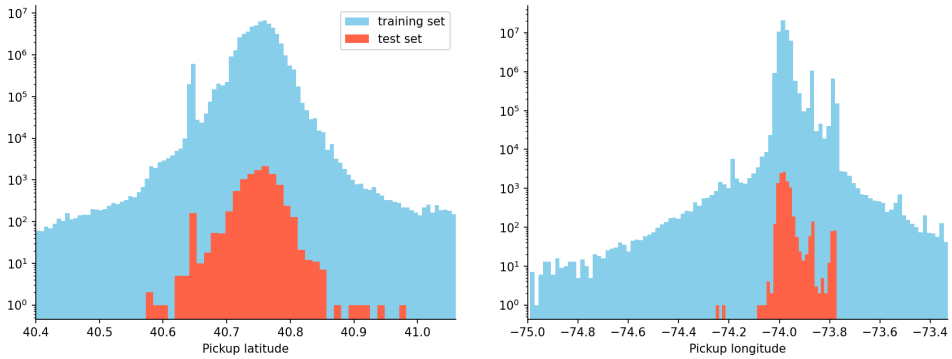
1. \$2.50 $\leq$ `fare_amout` $\leq$ \$800
2. 1 $\leq$ `passenger_count` $\leq$ 6
3. `year` $\geq$ 2009
4. 40.40 $\leq$ latitudes $\leq$ 41.06, $-75.00 \leq$ longitudes ≤ -73.33.

After all these truncations, we redo some of the figures plotted above. Any previous plots (with partial truncations) not discussed in this subsection are qualitatively the same as before the *full* truncation. The main differences lie in the plots about geographical data.

In Figure 11, we made the histograms of the pickup and dropoff coordinates, of the truncated training set and the test set. First, the test and training sets follow roughly the same distribution. Second, we notice slight peaks at latitude ≈ 40.65, and at longitudes $\approx -73.8, -74.2$. Incidentally, these are roughly the coordinates of the three airports in the greater New York area[1].

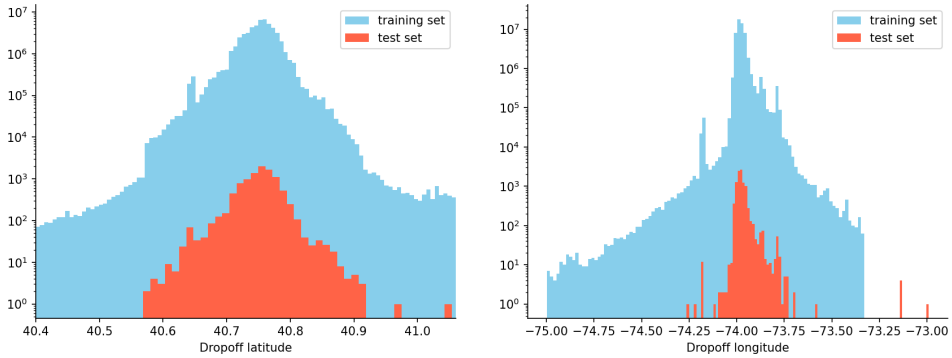
The Euclidean distance plot of Figure 9 now becomes Figure 12, in which we also plotted the test set data. Again, the two datasets give similar behaviour. The bumps at around 60 and 75 miles we previously observed were also gone, which makes sense because, as we remarked in the beginning, the farthest driving distance within NYC is only about 45 miles. The much narrower and smaller peak at around 10 miles is still present; the *driving* distance between the airports and Manhattan is about 10-15 miles.

[1] Coordinates of the airports in the greater New York area are roughly: EWR: (40.68, -74.175), JFK: (40.64, -73.78), LGA: (40.77, -73.87).



(a) Pickup latitudes

(b) Pickup longitudes



(c) Dropoff latitudes

(d) Dropoff longitudes

Figure 11: Pickup/dropoff coordinate histograms from the fully truncated training set and the test set.

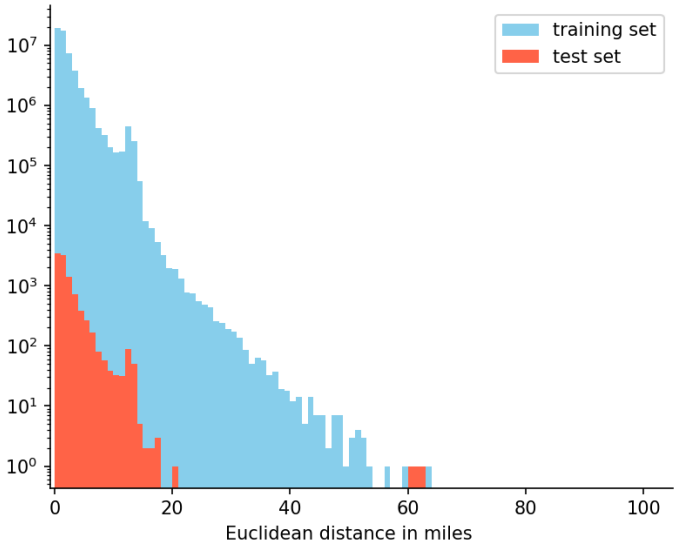


Figure 12: Euclidean distance between pickup and dropoff points from the fully truncated training set and the test set.

All in all, our training and test datasets have similar statistics.

2.6 Correlation Matrix

Lastly, we plot the correlation matrix of the fully truncated training set for a select few features (due to limited space). Expectedly, the the target `fare_amount` is strongly correlated with `euclidean_distance_miles`, and also with the coordinates.

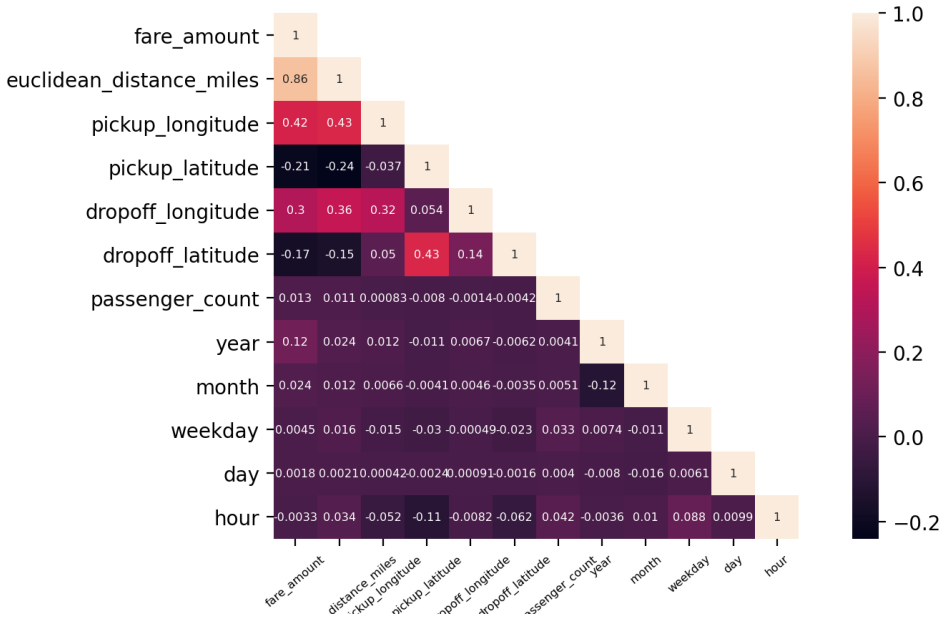


Figure 13: Correlation matrix from the fully truncated training.

3 Feature Pre-processing and Engineering

We now describe the feature pre-processing and engineering procedure, implemented in `pre-processing.py`.

3.1 Workflow

We first describe our workflow, where the starting point is the datetime-processed training set.

1. Load datetime-processed training set containing `year`, `month`, `weekday`, `day`, `hour` extracted from parsed datetime
2. Diacard outliers according to Section 2.5
3. Separate features and labels
4. Train-validation split: we use a 99.5-0.5 split; the validation set has ~270k instances

5. Compute Euclidean distances and bearings
6. Reverse geocoding: Extract zip codes and boroughs from the coordinates, and identify which coordinates fall in the neighbourhoods of the 3 airports
7. Encoding categorical variables
8. Select feature
9. Repeat Steps 5-8 for the datetime-processed test set.

Note that we do not perform a feature rescaling, because a) the training set is too large, and b) we anticipate using tree-based models which do not require such rescaling.

3.2 Euclidean Distances and Bearings

Throughout this work, we do not compute the *driving* distances from the coordinates. This can actually be done with the use of the `osmnx` and `networkx` packages (and included in `pre-processing.py`). We did not implement it simply due to the slow speed, considering the large size of our datasets.

We compute the Euclidean geodesic distances and the latitudinal/longitudinal displacements instead; within the city, it is very much the same as the Haversine distances. It is computed using simple trigonometry (remembering to convert degree to radian). The bearings are computed using `np.arctan2`, from which we extract its sines and cosines as features; the latter are periodic which make for better features.

3.3 Reverse Geocoding

By reverse geocoding, we specifically mean extracting the zip codes and boroughs (counties) from the coordinates. To this end, we import the “ZIP Code Tabulation Areas" dataset from the US Census Gazateer which contains zip code coordinates in the US, and a dataset from NYC Open Data that contains a map from NYC zip codes to counties. These two datasets are first merged. Then, using `sklearn.neighbors.KDTree` for its high speed, we extract the zip codes and counties from the coordinates. Furthermore, from this we also identify instances which begin/end at the neighbourhoods (taking into account the non-exactness of `KDTree`) of one of the airports, as well as those which begin and end in the same county.

All in all, the geographical new features are `pickup_zipcode`, `pickup_county`, `dropoff_zipcode`, `dropoff_county`, `pickup_airport`, `pickup_airport` and `dropoff_pickup_same_county`. The latter three are Boolean.

3.4 Encoding Categorical Variables

At this point we have the following categorical variables: `passenger_count`, `pickup_county`, `dropoff_county`, `weekday`, `year`, `month`, `day`, `hour`, `pickup_zipcode` and `dropoff_zipcode`. Inspired by the analysis from the last section, we perform one-hot encoding on `passenger_count`, `pickup_county` and `dropoff_county`, frequency encoding on `weekday` using `category_encoders`, and (target) mean encoding on `year`, `month`, `day`, `hour`, `pickup_zipcode` and `dropoff_zipcode`.

3.5 Feature Selection

We select the following features:

```
['pickup_longitude', 'pickup_latitude', 'dropoff_longitude',
'dropoff_latitude', 'daylight_saving', 'long_displacement',
'lat_displacement', 'euclidean_distance_miles',
'sin_direction', 'cos_direction', 'dropoff_pickup_same_county',
'pickup_airport', 'dropoff_airport', 'passenger_count_1',
'passenger_count_2', 'passenger_count_3', 'passenger_count_4',
'passenger_count_5', 'passenger_count_6', 'pickup_county_Bronx',
'pickup_county_Kings', 'pickup_county_New York', 'pickup_county_Queens',
'pickup_county_Richmond', 'pickup_county_not_in_NYC',
'dropoff_county_Bronx', 'dropoff_county_Kings',
'dropoff_county_New York', 'dropoff_county_Queens',
'dropoff_county_Richmond', 'dropoff_county_not_in_NYC',
'weekday_freq_encoded', 'year_mean_encoded', 'month_mean_encoded',
'day_mean_encoded', 'hour_mean_encoded', 'pickup_zipcode_mean_encoded',
'dropoff_zipcode_mean_encoded']
```

Lastly, we downcast some of the features where appropriate.

4 Models

Given the large size of the training set, we use `xgboost.XGBRegressor` and `lightgbm.LGBMRegressor`, hoping that everything can fit into the 16GB RAM of the local machine. It turns out it cannot. As a compromise, we only use part of the training (the “train" after the train-validation split) set to train the models. In principle, we should select the best model based on the lowest validation RMSE. But given the nature of Kaggle competition, we simply training them and submit all the predictions.

`xgboost.XGBRegressor` turns out to be more resource-hungry. We use 5 million instances from the training set, with the following hyperparameters:

```
XGBR_model = XGBRegressor(eval_metric = "rmse",
                          learning_rate = 0.05,
                          max_depth = 8,
                          n_estimators = 100,
                          reg_lambda = 0.7,
                          n_jobs = 6)
```

This results in an RMSE of 3.22539 on the test set, as verified by submitting to Kaggle.

For `lightgbm.LGBMRegressor` we use the following hyperparameters

```
LGBMreg = lightgbm.LGBMRegressor(boosting_type = 'gbdt',
                            learning_rate = 0.02,
                            num_leaves = 800,
                            n_estimators = 500,
                            num_iterations = 5000,
                            max_bin = 500,
                            feature_fraction = 0.7,
                            bagging_fraction = 0.7,
                            lambda_l2 = 0.5,
                            max_depth = 25,
```

```
                    silent = False)
```

With 5 million training instances, it gives an RMSE of 2.99131 on the test set, and it is significantly faster and less resource-hogging than `xgboost.XGBRegressor`. (We could try to tune the hyperparameters of the latter, but the time commitment is too high.) With 20 million training instances, the RMSE improved slightly to 2.96150. Any more training instances maxed out the RAM and caused a black screen of death.

Our choices of hyperparameters were inspired by our work on a previous project, which is another Kaggle competition predicting NYC taxi trip duration, given similar features as this project.

5 Predictions

Using `lightgbm.LGBMRegressor` trained with 20 million training instances, the test set predictions take the following distribution:

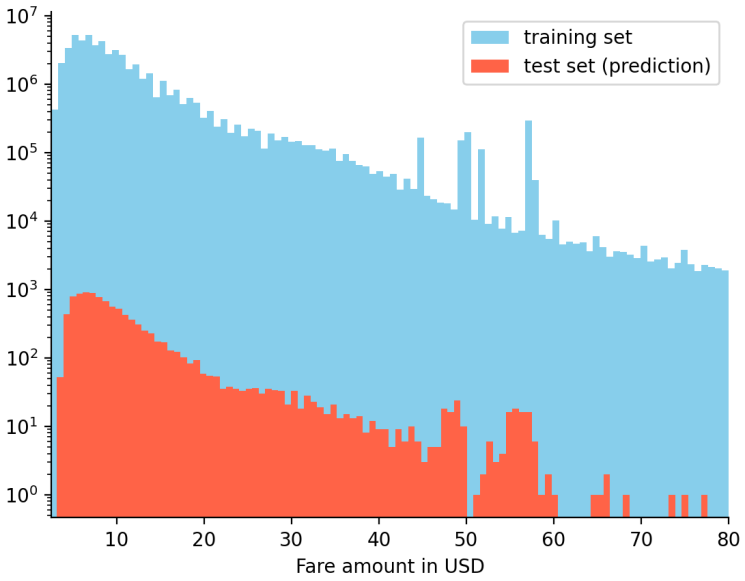


Figure 14: Test set predictions (red) from `lightgbm.LGBMRegressor` trained with 20 million training instances. The training set (blue) is the fully truncated set before the train-validation split.

This shows that the predictions take a very similar distribution as the full truncated training set. They both exhibit slight bumps at $\sim$ \$55, which could be due to the flat rate between the airports and Manhattan.

The `lightgbm.LGBMRegressor` model gives the following feature importances:

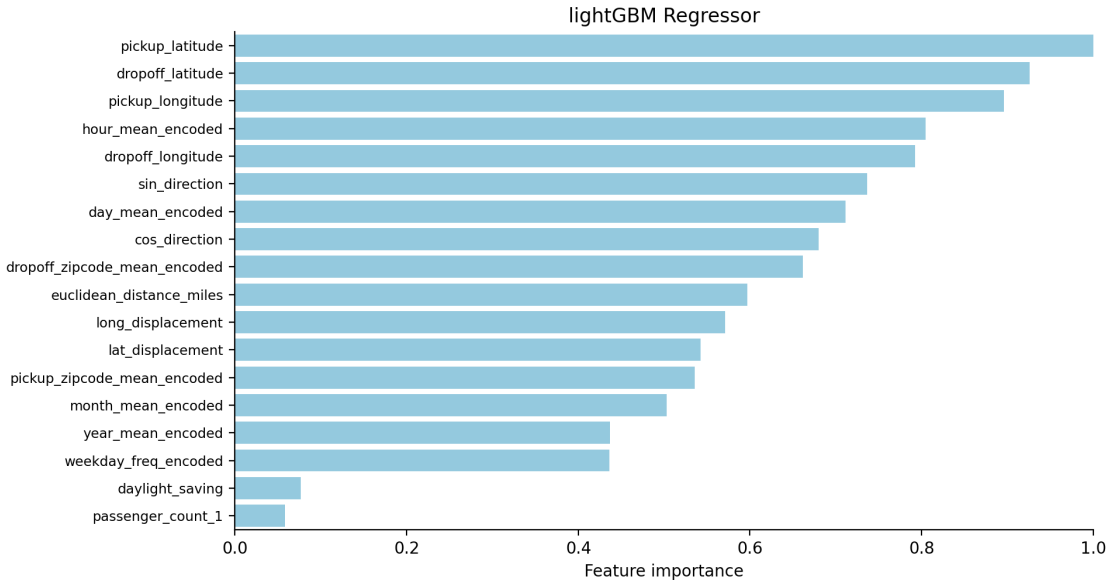


Figure 15: Feature importances of `lightgbm.LGBMRegressor` trained with 20 million training instances, normalised to one. We discarded some features with low importances in this plot.

This agrees with out intuition: geographical data strongly correlate with the taxi fare. The `hour` of the day also ranked among the highest, which could due to the rush hour/overnight surcharges. The `month`, `year` and `weekday` are also somewhat important, which may be due to holiday seasons, improving economy during 2009-2015, people rushing to work during the weekdays, etc.

6 Future Work

We have missed out a few elements that can improve our performance. First is extracting the driving distances. I am not sure how it can be done efficiently, but we can keep looking. Second, we should seek to implement `dask` to use the whole training set for model training. Third, we can further tune the hyperparameters to improve model performance.

7 Acknowledgment

We acknowledge the authors from various sources online, whose tools and techniques were borrowed and implemented in our codes. (I didn't keep track of the references.)