Advanced Python Final Report Group 5 — Predicting NYC Taxi Fare

Rui Chen `rc4753@nyu.edu`
Qiwenjing Jiang `qj336@nyu.edu`
Erqian Wang `ew1708@nyu.edu`
Jialing Li `jl9716@nyu.edu`
Zhongyan Wang `zw1462@nyu.edu`
Kristine Zeng `yz4792@nyu.edu`

Abstract

In this report, we present an investigation into predicting New York City taxi fares based on pickup and dropoff locations. We begin by developing a prediction pipeline utilizing traditional techniques, followed by an in-depth analysis and exploration to identify opportunities for improvement and optimization. Leveraging advanced libraries and frameworks acquired from our Advanced Python coursework, we successfully enhance the performance and efficiency of the prediction model. Overall, the achieved RMSE score is competitive on Kaggle leaderboard.

1 Introduction

In this report, we begin by providing an overview of the dataset utilized for predicting New York City taxi fares based on pickup and dropoff locations. We detail the data cleaning process, elaborating on the rationale behind our chosen approach. Subsequently, we delve into the feature engineering process to enhance the predictive capabilities of our models.

Considering the substantial size of the dataset, we adopt optimization techniques to ensure the efficient execution of our code. We then fit multiple machine learning models to the data and perform hyperparameter tuning to achieve optimal results.

2 Dataset and Data Loading

The dataset comprises 55 million records, capturing taxi trips in New York City from 2009 to 2015. Each entry provides crucial information such as pickup and dropoff locations in longitude and latitude, the number of passengers, pickup date, and fare amount. Our objective is to predict the fare amount for each trip using this comprehensive dataset. The size for the raw data is around 5.5 GB.

Given the larger size of data, Google Colab cannot take in the entire data in the RAM. We have to load the data to our local PC. Loading data using Pandas takes more than 5 minutes and often encounters kernel crush. We adopted a library called **Polars** that natively parallelizes the processing across the cores of the computer, which speeds up the data loading. Finally, we used **Datatable**'s `fread()` method to read in the data, which only takes less than 30 seconds.

3 Data Cleaning

During the data cleaning process, our primary focus is to eliminate records with missing or unreasonable values. We begin by examining the longitude and latitude of New York City to filter out entries that fall significantly outside the boundaries of the city's five boroughs. Additionally, considering the

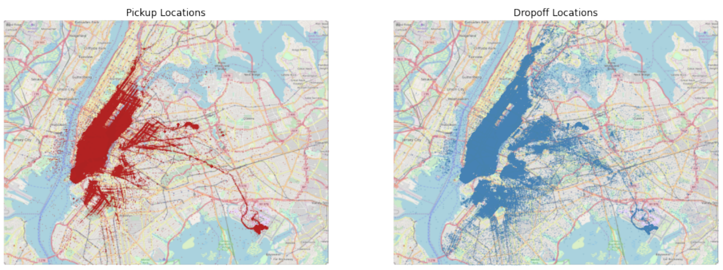

Figure 1: Locations on Map Visualization

maximum capacity of a NYC taxi is six passengers (including vans), we remove records with more than six passengers. Furthermore, we exclude data with fare amounts exceeding $2,000, as such instances are both uncommon and improbable. After the filtering process, we eliminated approximately 3 millions of rows, which leaves around 52 millions of rows of data.

4 Feature Engineering and Optimization

4.1 Vanilla Approach

A critical feature for our modeling is the calculation of the travel distance between the pickup and dropoff locations, as it serves as the primary factor influencing the fare amount. To achieve this, we utilize the **Geopy** package to determine the starting and ending points based on their longitude and latitude coordinates. Subsequently, we employ a function `geodesic()` to accurately compute the distance traveled between these two points. To carry out this task, we employed a lambda function for each row of data. However, as the Geopy function needs to be called for every individual record, the computation time proved to be extensive. The entire calculation process spanned over 10 hours, prompting us to explore alternative approaches in order to enhance calculation efficiency.

4.2 Optimization Using MPI & Itertools

We first parallelize the computation using 8 processes in MPI. Because each buffer can only hold 2GB data, we have to split the data in 6 chunks. Each computation reads in 10 millions of data and we use `starmap()` and `list(zip())` to speed up the distance computation. After we have the resulting dataframe for each chunk, we then concatenate the 6 chunks to get the entire dataset. Below is a snippet of our code.

```
pickup = list(zip(sub.pickup_latitude, sub.pickup_longitude))
dropoff = list(zip(sub.dropoff_latitude, sub.dropoff_longitude))
pairs = list(zip(pickup, dropoff))
sub['dist_miles'] = list(starmap(lambda u, v: geodesic(u, v).miles, pairs))
```

The entire process takes less than one hour, but we still need to repeat the calculation for 6 times, which makes us want to explore more ways to improve the calculation.

4.3 Optimization Using Haversine Formula

After looking into the documentation of Geopy, we discovered that their calculation of distance is essentially based on Haversine Formula.

Given two points $p_1 = (\phi_1, \lambda_1), p_2 = (\phi_2, \lambda_2)$, where ϕ refers to latitudes and λ refers to longitude (in radians), the distance between p_1 and p_2 on a sphere is defined as follows.

$$d = 2R \times \arcsin\left(\sqrt{\sin^2\left(\frac{\phi_2 - \phi_1}{2}\right) + \cos\phi_1 \cdot \cos\phi_2 \cdot \sin^2\left(\frac{\lambda_2 - \lambda_1}{2}\right)}\right),$$

where $R = 3958.8$ is the Earth radius in miles.

Consequently, we devised a custom function derived from the formula and applied it to the entire dataframe. Remarkably, the computation time was reduced to less than 5 seconds. The results generated by our custom function closely approximated those calculated using Geopy. Owing to the significant improvement in processing time, we opted to employ our customized function for distance calculations.

4.4 Date and Airport Flat Rate

In addition to the distance feature, we extracted the year, month, week, day of the week, and hour from the original pickup time attribute. Subsequently, we employed the `OneHotEncoder()` function to create dummy variables for these extracted features.

According to NYC.gov, there are three airports (LGA, JFK, EWR) in the vicinity of New York City, with trips to and from these airports being charged differently compared to standard metered fares. Utilizing the haversine formula, we computed the distances between the pickup and dropoff locations and the airports, enabling us to create airport-related features for our model.

5 Models

5.1 Model Selection and Hyper-Parameter Tuning

- **Linear models**: We first fit a linear regression model as our baseline. The validation RMSE is 4.4266. Then, to add regularization, we fit both LASSO and Ridge regression with different alpha. The linear model is Ridge regression woth $\alpha = 0.01$.

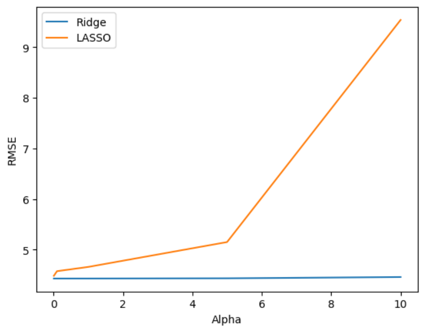

Figure 2: Regularized Model Performance Comparison Across Different Alpha

- **XGBoost**: We then fit the XGBoost model to get better performance as it is a highly efficient ensemble tree method. With `num_estimator = 500` and `max_depth = 5`, the RMSE = 3.3426. However, the fitting time is long, one set of training takes more than 15 minutes. After adding the additional airport feature, we tune the hyper-parameters.

# of estimator	max depth	RMSE
10	500	3.284
10	1000	3.299
12	500	3.321

- **LightGBM**: Light Gradient Boosting Machine's efficiency in training and memory usage, along with its better handling of categorical features and scalability, make it an attractive option for our prediction task. We fit the model with and without the airport flat rate feature, and tuning the model for different hyper-parameters.

# of boost round	RMSE w/o airport	RMSE w/ airport
1000	3.333	3.262
5000	3.244	3.187
10000	3.222	3.167

After an extensive hyper-parameter tuning process, our best hyper-parameter combination is `num_leaves: 1000`, `learning_rate: 0.05`, `min_gain_to_split: 0.5`, `min_data_in_leaf: 30`, and RMSE is **3.1137**.

In conclusion, we compared the performance of various linear regression models with and without regularization, XGBoost, and LightGBM for our regression task. After conducting a thorough evaluation, LightGBM emerged as the top-performing model, delivering the lowest RMSE and the shortest training time among all considered models. Furthermore, incorporating the 'airport' features into our models led to a noticeable decrease in RMSE, suggesting that this additional feature enhances the predictive accuracy of our chosen model. Overall, LightGBM's superior performance and efficiency make it the optimal choice for our prediction task.

5.2 Feature Importance

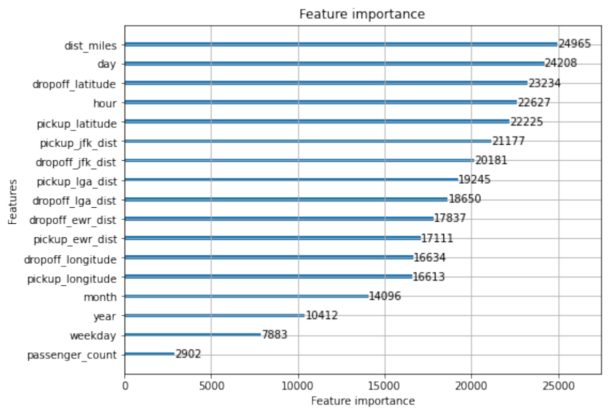

Figure 3: Feature Importance of LightGBM

In our analysis of feature importance using the top-performing model, we identified several key factors influencing New York City taxi fares. As illustrated in the accompanying graph, the most significant feature is the distance traveled. This comes as no surprise, given that NYC taxi fares are primarily based on distance traveled.

In addition to geo-location features, which are already encoded as distance, the day and hour emerged as crucial factors. These attributes have a considerable impact on fares due to their influence on traffic congestion during peak hours. In addition, airport features are also deciding factors for fare, as they are metered differently from normal trips.

6 Conclusion

To accurately predict NYC taxi fares, we implemented a comprehensive process that includes data cleaning, feature engineering, model selection, and hyperparameter tuning. Considering the vast size of the dataset, we optimized our code to enhance its efficiency. This optimization process involved a delicate balance between the code's efficiency and its accuracy. Ultimately, we achieved exceptional results in our fare prediction model.

References

[1] Kaggle. (2018). *New York City Taxi Fare Prediction* https://www.kaggle.com/competitions/new-york-city-taxi-fare-prediction/data

[2] NYC Government. (2023). *Taxi Fare* https://www.nyc.gov/site/tlc/passengers/taxi-fare.page

[3] Wikipedia. (2021). *Haversine formula* https://en.wikipedia.org/wiki/Haversine-formula