Manhattan Taxi Tip Amount Analysis and Prediction Project Report

Chenxin Gu cg3423@nyu.edu

Junwen Fang jf3994@nyu.edu

Adam Xu tx543@nyu.edu

Zipei Wang zw2458@nyu.edu

Ziyu Qi zq2127@nyu.edu

May 6, 2025

1 Problem Setup

Tipping behavior in urban transportation remains underexplored despite its economic relevance. NYC taxi services provide a rich setting to examine how temporal, spatial, and contextual factors influence gratuity decisions. Unlike fixed fares, tips vary based on factors such as trip timing, distance, location, surcharges, and payment methods. Modeling tip amounts can improve passenger experience, guide driver strategies, and support operations. This project uses NYC TLC trip-level data to predict tips and uncover underlying patterns.

1.1 Allocation of Work

In this project, Adam, Zipei and Ziyu are responsible for data preprocessing. Chenxin and Junwen are responsible for model and optimization. Code for our project is turned in seperately in the attached zip file or through github: https://github.com/EllieWzp/advanced_python_final.

2 Data Preprocessing

Our original dataset was obtained from the 2024 public NYC Taxi & Limousine Commission (TLC) trip record dataset, which includes more than 3,600,000 detailed trip-level information such as timestamps, locations, fare components amounts.

2.1 Data Cleaning and Filtering

Rows with missing values in key fields such as `fare_amount`, `tip_amount`, and `trip_distance` were removed. Based on exploratory analysis, we excluded trips with zero tip amounts, which is common in the dataset but unexplainable without demographic context, as well as those exceeding the 98th percentile to mitigate the effect of outliers. These filtering steps allowed us to train the model on meaningful, non-extreme tipping behavior. The resulting dataset included 15 features spanning numerical and contextual attributes.

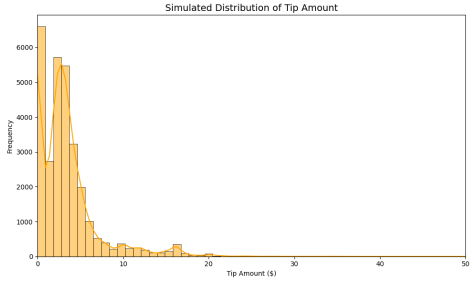

Figure 1: Distribution of tip amounts before filtering.

To ensure more stable model training, we removed trips with a tip amount of 0 and excluded the top

2% of outliers. The motivation behind this is two-fold: (1) zero tips often correspond to noise, short rides, or payment methods like cash where tips aren't recorded; (2) the top 2% of extreme values can distort both the loss function and model interpretation due to their disproportionately large influence, as shown in Figure 1.

2.2 Feature Transformation

All numerical features were standardized using `StandardScaler` to ensure consistent scaling across predictors. Categorical fields were converted into binary indicators to reflect relevant domain knowledge. TLC zone IDs for pickup and dropoff were grouped into four macro-regions—Midtown, Uptown, Downtown, and Other. The `VendorID` field was encoded as a binary variable, as only two vendor types were recorded. Time-based features were converted into binary flags indicating whether a trip occurred during weekends, peak weekday hours (4:00–8:00 PM, Monday to Friday), or night hours (8:00 PM–6:00 AM), in accordance with NYC's official surcharge policies.

2.3 Feature Selection

To reduce redundancy and improve interpretability, we applied three key steps: (1) Highly correlated variables were dropped, including `mta_tax` (which is correlated with `improvement_surcharge`, $\rho = 0.85$) and `total_amount`, which includes the target. (2) Low-variance categorical features such as `RatecodeID` were removed due to limited predictive value. (3) Lasso regression was used to select the most informative predictors, including `fare_amount`, `congestion_surcharge`, `tolls_amount`, and `airport_fee`. These preprocessing steps ensured a clean, interpretable, and well-scaled dataset suitable for downstream modeling.

3 Model Selection and Baseline Performance

3.1 Model Selection: XGBoost

We began by evaluating several baseline models including Ordinary Least Squares (OLS) regression and Random Forests. Both exhibited limited predictive performance, failing to capture the non-linear structure of the data. After removing zero-tip entries and the top 2% outliers, OLS achieved an R^2 of only 0.42. In contrast, **XGBoost** demonstrated superior performance with minimal tuning, achieving an R^2 of approximately 0.68 on the same cleaned dataset, therefore was selected as the final modeling framework for further optimization and evaluation.

3.2 Objective function

We then evaluated several hand-written loss functions in XGBoost:

- **Squared Error (SE)**: Penalizes larger errors quadratically. It consistently performed well with RMSE ≈ 1.60 and $R^2 \approx 0.69$, and yielded short training times.

$$L(y_{\text{pred}}, y_{\text{true}}) = (y_{\text{pred}} - y_{\text{true}})^2$$

- **Squared Log Error (SLE)**: Reduces sensitivity to large values and suits right-skewed data. It achieved RMSE ≈ 1.66 and $R^2 \approx 0.66$ with similar efficiency.

$$L(y_{\text{pred}}, y_{\text{true}}) = (\log(1 + y_{\text{pred}}) - \log(1 + y_{\text{true}}))^2$$

- **Absolute Error (AE)**: More robust to outliers but non-differentiable at zero. It showed poor performance with RMSE > 3 and $R^2 \approx 0$ in early experiments.

$$L(y_{\text{pred}}, y_{\text{true}}) = |y_{\text{pred}} - y_{\text{true}}|$$

- **Gamma Negative Log-Likelihood (NLL)**: Designed for positive, skewed targets, but yielded similarly poor results as AE.

$$L(y_{\text{pred}}, y_{\text{true}}) = \log(y_{\text{pred}}) + \frac{y_{\text{true}}}{y_{\text{pred}}}$$

3.3 Baseline Performance

To benchmark optimization efficiency, we implemented the objective functions `reg:squarederror` and `reg:squaredlogerror` by hand and trained XGBoost without any tuning. For `reg:squarederror`, the hand-written implementation yielded a baseline performance of $R^2 \approx 0.69$ and an RMSE around 1.60. For `reg:squaredlogerror`, performance was slightly lower, with $R^2 \approx 0.66$ and RMSE around 1.66.

The total training time was relatively long, taking approximately 81.14 seconds for `reg:squarederror` and 97.46 seconds for `reg:squaredlogerror`. These results served as our baseline reference before applying any optimization strategies.

4 Optimization

In this section, we discuss optimization both performance-wise and time-wise. Beyond simply improving R^2 scores, our goal was to design an efficient training pipeline that could scale to larger datasets or more frequent retraining. We carefully examined various search strategies and algorithmic enhancements to reduce computational overhead without sacrificing model accuracy.

4.1 Hyperparameter Search

We began with a brute-force grid search over all 125 combinations of `max_depth`, `learning_rate`, and `n_estimators` to explore which hyperparameters best fit our dataset. This provides us comprehensive coverage but at high computational cost.

- `max_depth`: {4, 5, 6, 7, 8}
- `learning_rate`: {0.01, 0.02, 0.05, 0.1, 0.2}
- `n_estimators`: {100, 200, 300, 400, 800}

This resulted in 125 configurations per objective function, providing comprehensive coverage but at high computational cost.

4.2 Speed-Up Techniques

We benchmarked several optimization strategies to reduce training time while maintaining model performance. We tested all algorithms on a 10% sample of the entire dataset.

- **Switching from Hand-Written Loops to Inbuilt Methods.** Our initial implementation manually iterated over hyperparameter combinations. Replacing this with XGBoost's inbuilt API for training and prediction significantly reduced overhead—cutting total runtime from 81.14 to 72.32 seconds for `reg:squarederror`, and from 97.46 to 91.22 seconds for `reg:squaredlogerror`. This gain primarily came from avoiding redundant Python loops and leveraging optimized C++ routines under the hood.
- **Random Search.** Instead of exhaustively evaluating all 125 parameter combinations, we randomly sampled 20. Based on logs from the full grid search, we observed that many parameter combinations led to similar performance. This suggested that a large portion of the grid was redundant, making random sampling a more efficient linear-time strategy. This alone lowered runtime to 13.43 and 14.44 seconds respectively.
- **Histogram-based Tree Construction.** Enabling the `tree_method=hist` flag allowed XGBoost to use histogram binning for faster split finding. With this change, total training time dropped further to under 9 and 12 seconds.
- **Sampling via `itertools`.** To combine reproducibility with flexibility, we used `itertools.product` to generate the full hyperparameter grid and sampled from it directly. Adding `subsample=0.8` and `colsample_bytree=0.8` for additional regularization, final runtimes reached 7.08 and 9.85 seconds.

Optimization Step	reg:squarederror (s)	reg:squaredlogerror (s)
Hand-written Implementation	81.14	97.46
Inbuilt (Baseline)	72.32	91.22
+ Random Search	13.43	14.44
+ Hist Method	8.82	11.52
+itertools Sampling	**7.08**	**9.85**

Table 1: Training time (in seconds) for different optimization strategies and loss functions.

Table 1 and figure 2 summarize the cumulative effects of our optimization steps. Overall, these strategies reduced training time by over 90% compared to the original hand-written baseline, without compromising predictive performance.

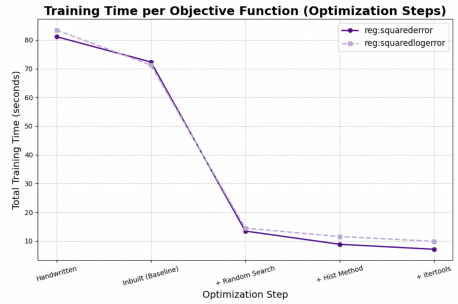

Figure 2: Training Time Per Objective Function

5 Conclusion

5.1 Execution on Full Dataset

We deployed our final XGBoost model on the full-scale dataset, which was approximately 10 times larger than the development subset. As before, we filtered out zero-tip entries and excluded the top 2% of extreme values to ensure consistent training conditions. The model utilized 15 features, including binary time and location indicators (e.g., `is_peak_hour`, `PU_is_midtown`) as well as numerical variables such as `fare_amount`, `trip_distance`, and `congestion_surcharge`.

Hyperparameters were selected based on prior tuning: we used `max_depth=7`, `learning_rate=0.02`, and `n_estimators=300`, with `subsample=0.8`, `colsample_bytree=0.8`, and `tree_method=hist` for efficiency. `Early_stopping_rounds=20` was applied to prevent overfitting.

On the larger dataset, performance improved: the `reg:squarederror` objective achieved $R^2 = 0.7231$ and RMSE = 1.5161, while `reg:squaredlogerror` yielded $R^2 = 0.7098$ and RMSE = 1.5521. Average training time per trial remained efficient at just 4.57 seconds.

Overall, the model scaled well to the full dataset, delivering better predictive accuracy and maintaining computational efficiency. These results suggest that fare-related values and temporal factors remain robust signals for understanding and predicting taxi tipping behavior in New York City.

5.2 Future Work

While our model performed well, several extensions remain. Adding features like holidays or weather may boost accuracy, and reframing the task as a classification problem (e.g., whether a tip is given) could offer new insights. Tools like SHAP can improve interpretability. As tipping rises in recent years, such models may also serve as fair benchmarks for passengers, reducing pressure while acknowledging that zero-tip choices still occur.

6 Reference

New York City Taxi and Limousine Commission (TLC). *TLC Trip Record Data (2021)*. Available at: https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page