New York City Taxi Trip Duration Prediction Project Report

Baher Alabbar*

July 28, 2025

Contents

1 Introduction

This project focuses on predicting taxi trip durations in New York City. The main goal was to learn the entire machine learning process from start to finish — including data analysis, feature selection, modeling, and deployment. I began with exploratory data analysis (EDA), then improved the model by selecting important features and refining it in Python. The initial model had an R^2 score of 0.12, which improved to 0.69 after feature engineering and data cleaning. Finally, I created an API to simulate deploying the model and integrated it into a command-line app to understand how deployment and application work together.

2 EDA Insights & Feature Discovery

This section summarizes the most important insights from the Exploratory Data Analysis (EDA) that directly helped with building the model. For all the detailed analysis and visualizations, please refer to the full EDA notebook included in the project.

2.1 Data Overview and Cleaning

The original dataset contained the following 10 columns:

- `id`
- `vendor_id`
- `pickup_datetime`
- `passenger_count`
- `pickup_longitude`, `pickup_latitude`
- `dropoff_longitude`, `dropoff_latitude`
- `store_and_fwd_flag`
- `trip_duration` (target variable)

Key preprocessing steps applied:

- `id` was immediately dropped as it held no useful information for modeling.
- `vendor_id`, `passenger_count`, and `store_and_fwd_flag` were treated as categorical variables.
- `pickup_datetime` was converted into a datetime object and later used to derive useful time-based features such as hour, day of week, and month.
- Rows with coordinates clearly falling outside of NYC boundaries were removed to maintain a consistent distribution.

2.2 Target Distribution

`trip_duration` was highly right-skewed. To reduce skewness and mitigate the effect of outliers, we applied a log transform: `log1p(trip_duration)`.

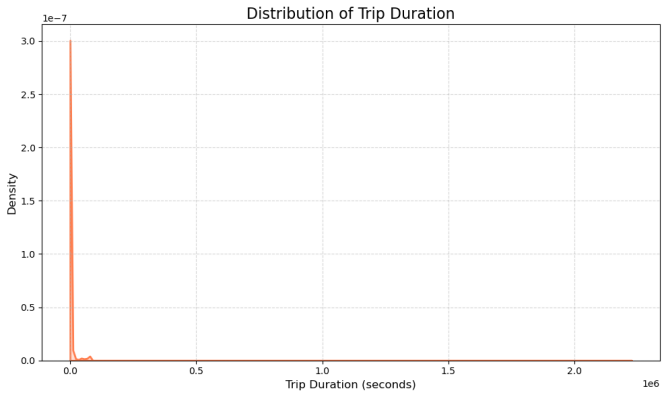

Figure 1: *Histogram of the original (raw)* ***`trip_duration`*** *showing right-skewed distribution with long tails.*

We also removed extreme outliers beyond a certain quantile threshold.

*author: baher.alabbar@gmail.com
Date: July 28, 2025

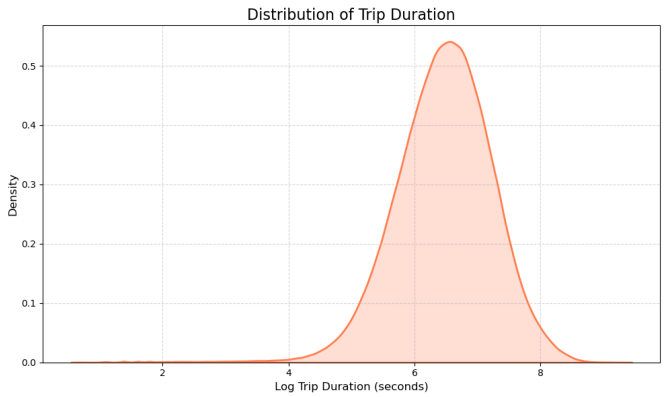

Figure 2: *Histogram of the `log1p`-transformed `trip_duration` showing reduced skewness and a more normal distribution.*

2.3 Store and Forward Flag ("Y" vs "N")

Trips with flag `"Y"` were hypothesized to have longer durations. This was validated by analyzing central tendency and skewness.

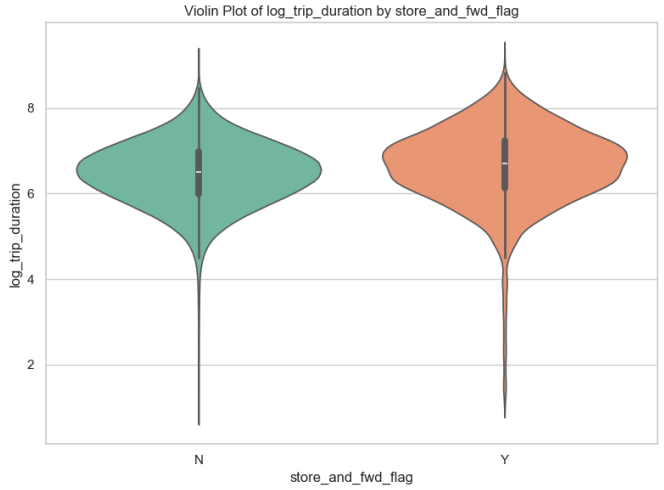

Figure 3: *Comparison of `log_trip_duration` distributions between `store_and_fwd_flag = "Y"` and `"N"`. The "Y" group shows a higher median and longer durations overall.*

Metric	"Y" Flag	"N" Flag
Mean (log) Duration	6.63	6.46
Median (log) Duration	6.70	6.49
Skewness	-1.22	-0.62

Table 1: *Comparison of trip duration statistics by `store_and_fwd_flag`*

This confirmed that `"Y"` trips tend to be longer, even after accounting for skewness.

2.4 Spatial Insights (Map-Based)

Plotting pickup and dropoff coordinates revealed that many of the longest trips originated or ended near airports (JFK, LaGuardia). Color gradients were used to reflect log-transformed trip duration.

Figure 4: *Pickup Locations Colored by `log_trip_duration`. Lighter points indicate longer trips. Note the cluster of long-duration trips near NYC airports.*

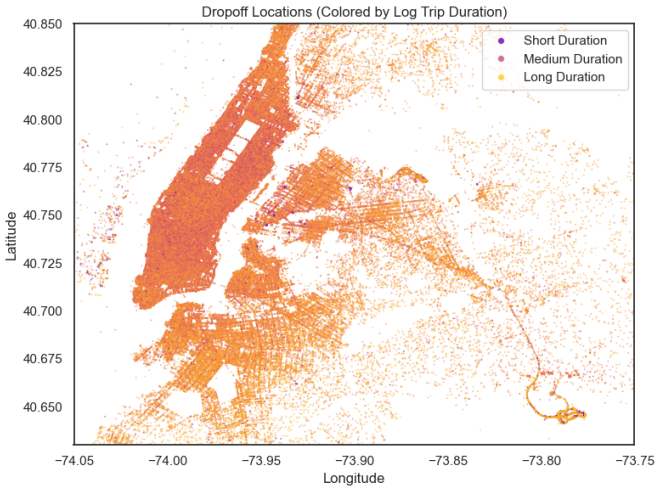

Figure 5: *Dropoff Locations Colored by `log_trip_duration`.*

We engineered two flags based on these insights:

- `is_jfk_airport`
- `is_lg_airport`

These flags are set to true when either pickup or drop off coordinates are inside the range of the airport coordinates.

Reference:

- JFK Airport Map
- LaGuardia Airport Map

2.5 Time-Based Patterns

A clear anomaly was observed in January due to the 2016 Blizzard, which significantly affected trip durations for a few days.

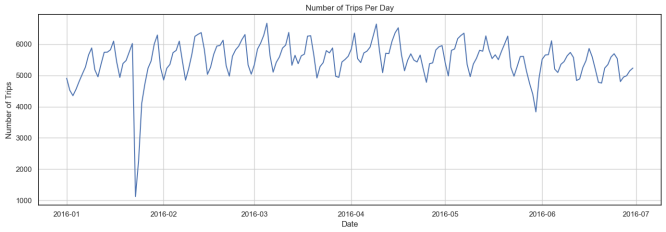

Figure 6: *Time series of daily trip counts. A significant drop is observed between January 22–24, likely due to the January 2016 U.S. blizzard.*

Affected rows were removed before modeling.
Additional time-related insights:

- **Rush Hours:** Spikes in trip count and duration were found between 6–9 AM and 5–7 PM.
- **Seasonality:** As shown in **Figure 7**, average durations were higher during summer and especially June, likely due to tourist behavior and longer airport routes.

These led to engineered features like:

- `is_rush_hour`
- `is_summer`

2.6 Passenger Count & Vendor Insights

Analyzing the distribution of passengers for `vendor_id = 1`, as shown in **Figure 8**, reveals that this vendor does not handle any trips with more than 4 passengers. This clearly implies that all high-passenger-count trips (i.e., more than 4 passengers) must be handled by `vendor_id = 2`, suggesting that this vendor likely operates larger vehicles.

Engineered Feature: `requires_large_vehicle` — This flag is set to `True` when the number of passengers exceeds 4.

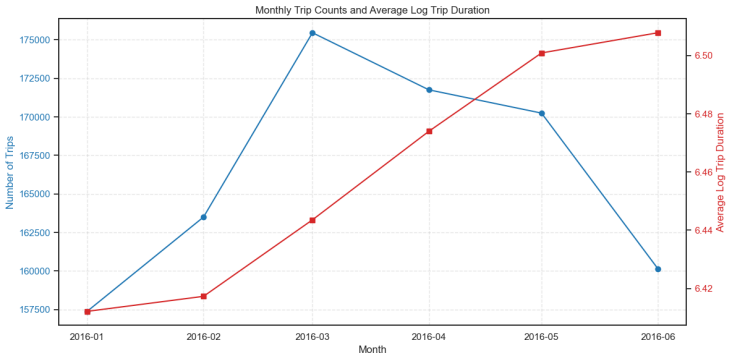

Figure 7: *Average log-transformed trip duration by month. The graph highlights higher average durations in January and during the summer months, which aligns with increased tourist activity and longer airport-related trips.*

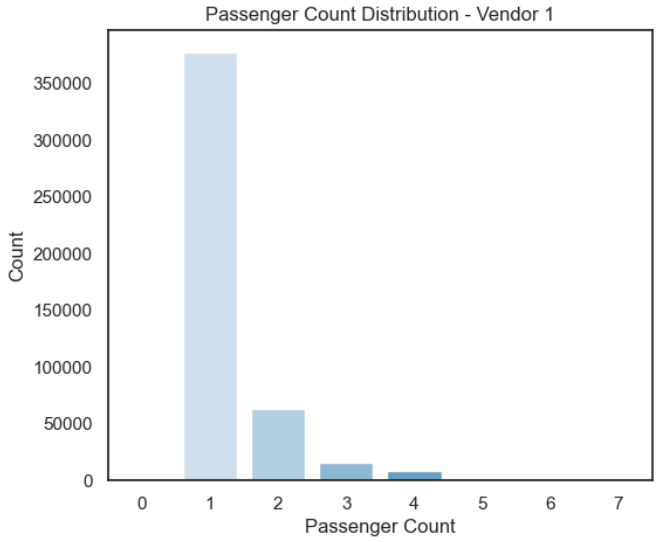

Figure 8: *Passenger count distribution for Vendor 1. Most trips involve 1–2 passengers, and no trips with more than 4 passengers are recorded.*

2.7 Distance Features

Trip distance was calculated from coordinates and transformed:

- `trip_distance`
- `log_trip_distance`, `trip_distance_cube`, `trip_distance_square`, etc.

These features, while simple, boosted performance significantly.

2.8 Virtual Speed & Virtual Time

To better approximate the trip duration using engineered features, we derived two synthetic variables: `virtual_speed` and `virtual_time`. These were inspired by the basic physics formula:

$$\text{time} = \frac{\text{distance}}{\text{speed}}$$

Since actual speed data is not available in the dataset, we designed a virtual speed that adjusts based on trip characteristics known to influence duration.

Intuition: Longer trips are often associated with certain factors — airport proximity, rush hour, summer traffic, and the presence of the `"Y"` store-and-forward flag. Each of these contributes to increased travel time, which in turn implies lower effective speed.

Virtual Speed Logic: We begin with a base speed of 32 units. For each condition that likely increases duration, we halve the virtual speed:

- `is_rush_hour`
- `is_summer`
- `is_jfk_airport` or `is_lg_airport`
- `store_and_fwd_flag = "Y"`

This results in a virtual speed that ranges from:

$$32 \rightarrow 2 \quad \text{(if all four conditions are true)}$$

Virtual Time Calculation: Once `virtual_speed` is computed for each trip, we calculate:

$$\texttt{virtual\_time} = \frac{\texttt{trip\_distance}}{\texttt{virtual\_speed}}$$

These engineered features encapsulate multiple known influences on trip duration into a single numeric signal. Notably, `virtual_time` was found to be more strongly correlated with the target than many original or individually engineered features.

2.9 Feature Correlation & Final Selection

Features were analyzed for correlation with `log_trip_duration`. While some low-correlation features were dropped, further testing revealed that even weakly correlated features (e.g., when one-hot encoded) could improve model performance.

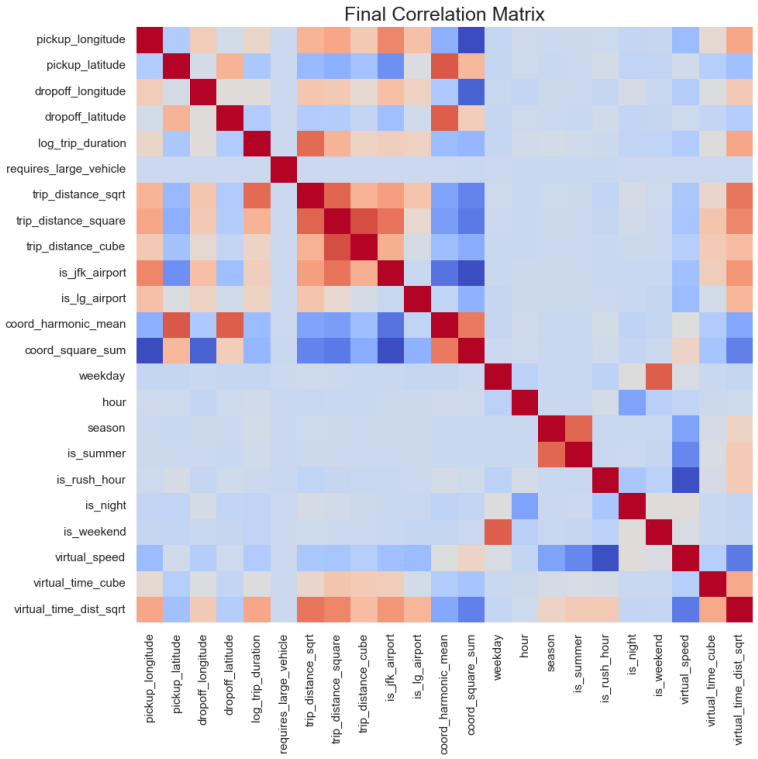

Figure 9: *Final correlation heatmap showing relationships between engineered features and the log-transformed trip duration. Strong positive correlation is observed with features like* ***`trip_distance`****,* ***`virtual_time`****, and airport-related flags.*

2.10 Lessons Learned from EDA

- EDA is not about plotting everything — it's about asking meaningful questions and testing assumptions.
- Good visualizations guided nearly every feature engineering decision made in this project.
- Correlation does not fully dictate importance — some features only reveal power after encoding or transformation.
- Final feature selection must balance domain knowledge, data patterns, and experimentation.

3 Modeling & Results

3.1 Modeling Objective

I used Ridge Regression (with $\alpha = 1$) as the base model to avoid overfitting due to the increasing number of engineered features. Unlike simple Linear Regression, Ridge penalizes large coefficients, offering a more stable solution. More complex models like XGBoost were avoided intentionally to keep the focus on understanding the true effect of feature engineering.

3.2 Feature Importance and Impact

This section analyzes how each engineered feature impacted model performance.

- `trip_distance` and its manipulations (log/square): Had the highest impact, increasing R^2 from 0.49 to 0.69 when included.
- `virtual_speed`: Decreased performance by 3 when used numerically, but improved by 7 when treated as categorical.
- `virtual_time`: Strongly correlated with target; useful.
- `is_rush_hour`, `is_summer`: Some impact, but already absorbed into virtual speed.
- `is_jfk_airport`, `is_lg_airport`: Slight impact, not as significant as expected.
- `store_and_fwd_flag`: No major performance boost, likely due to class imbalance (most values are "N"). Undersampling the "N" class to balance with the minority "Y" class could have potentially improved its impact, even though this might slightly affect the data distribution.
- `requires_large_vehicle`: Had minimal impact.

Interestingly, features like `month` and `minute` performed better when one-hot encoded. Also, applying log/square transformations—even when causing multicollinearity—improved model accuracy.

Key Insight: Some features you expect to be powerful may turn out useless, while unexpected ones help more than anticipated.

3.3 Model Comparison

Table 2: *Model Variants and R^2 Performance*

Model	Description	Train R^2	Val R^2
Baseline	Raw data only, no feature engineering	0.1200	0.1205
Model 1	Key engineered features	0.5244	0.5276
Model 2	No column transformations	0.6347	0.6355
Model 3	No one-hot encoding	0.6550	0.6552
Model 4	Removed outliers early	0.6383	0.6379
Model 5	Clean evaluation, no test leakage	0.6941	0.6947
Model 6	No scaling	0.6933	0.6934
Model 7	MinMax scaling	0.6901	0.6905
Final	Log target, one-hot, outliers after log	**0.6946**	**0.6949**

Table 3: *Model Insights and Observations*

Model	Notes
Baseline	Establishes starting point to measure impact of feature engineering.
Model 1	Feature engineering gave a large performance boost.
Model 2	Only log-transform on target used; raw inputs.
Model 3	Surprisingly strong without encoding; better with one-hot.
Model 4	Removing outliers early degraded performance slightly.
Model 5	Clean evaluation, best generalization result.
Model 6	No feature scaling slightly lowered performance.
Model 7	MinMax scaling was less effective than standardization.
Final	Best setup overall: log target, encoded categoricals, clean outlier handling.

3.4 Final Modeling Pipeline

The modeling pipeline included one-hot encoding and Ridge Regression with standardization, but did not include other preprocessing steps. See **Figure 10** for the complete pipeline.

Modeling Steps:

1. Fixing data types.
2. Outlier removal after log transform.
3. Feature engineering.
4. Column dropping.
5. Final encoding and scaling.

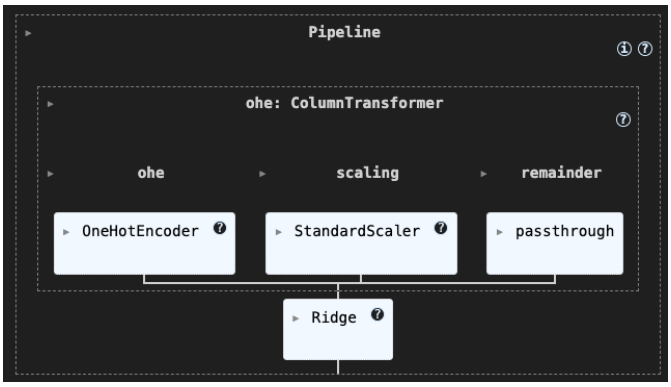

Figure 10: *Modeling pipeline diagram.*

3.5 Lessons Learned

- Feature engineering alone improved R^2 from 0.12 to 0.69.
- Removing features during EDA was a mistake—testing them all was more informative.
- Feature selection based on correlation alone can be misleading.
- Data leakage is a major concern (e.g., `store_and_fwd_flag`) and should be considered early.
- Even simple transformations like log/square boosted performance significantly.
- Choosing encoding types (e.g., one-hot) is just as important as the features themselves.
- Model tracking tools are essential for trying many variants systematically.

These lessons shaped my understanding of the modeling process and reinforced the need for iteration, testing, and careful validation. The importance of balancing feature thinking with automation tools and smart model selection became evident.

4 API & CLI Deployment

4.1 Inference Data Preparation

The `inference-input-prep.ipynb` notebook made the transition from model evaluation on a test set to real-time, single-trip inference much smoother. It focused entirely on identifying the necessary input fields for prediction and applying the same preprocessing logic used during training. As a result, this notebook served as a bridge between the modeling phase and the API deployment, allowing seamless integration of the final model into production.

4.2 FastAPI Service

FastAPI was chosen to develop a RESTful API for the deployed model. The API includes several endpoints, designed for both functional prediction and user guidance:

Table 4: *API Endpoint Summary*

Method	Endpoint	Description
POST	`/predict`	Predicts trip duration based on user input. Returns `{"duration": 437}`.
GET	`/features`	Lists all features required for prediction.
GET	`/features/sample`	Returns a sample JSON input to guide usage.
GET	`/about`	Provides information about the model and its purpose.
POST	`/validate`	Validates input format and schema.
GET	`/version`	Returns model and API version information.
GET	`/help`	Lists all endpoints with short descriptions.

Once the final model and pipeline were prepared, deploying them became a structured, streamlined process. From data preparation to endpoint creation, each step followed a clear, systematic path.

4.3 CLI Tool

To emulate API functionality via the terminal, a command-line interface (CLI) tool was developed using Python's `argparse` and `requests`. The tool allows users to send API requests through custom arguments. If the `-endpoint` is set to `predict` or `validate`, additional flags are used to pass trip details:

- `-vendor_id`, `-passenger_count`
- `-pickup_longitude`, `-pickup_latitude`
- `-dropoff_longitude`, `-dropoff_latitude`
- `-pickup_date`, `-pickup_time`
- `-store_and_fwd_flag`

This CLI was useful for testing and prototyping predictions without a web frontend.

4.4 Pipeline Advantages in Deployment

A key learning in this stage was the enormous benefit of building and saving the full modeling pipeline using `scikit-learn`'s `Pipeline`. This design allowed preprocessing and prediction to be encapsulated within a single object. Without this approach, every encoder, scaler, and transformation would have to be saved and reapplied individually.

Using a pipeline ensures:

- All encoders (e.g., `OneHotEncoder`) and scalers (e.g., `StandardScaler`) are trained on the training data only.
- No accidental data leakage from fitting encoders on validation or test sets.
- Clean integration during inference, reducing the chance of preprocessing mismatches.

This naturally led to one of the most critical realizations in the deployment phase: hidden data leakage.

4.5 Hidden Data Leakages and Lessons Learned

While building the inference logic, a significant issue arose: the model depended on the feature `store_and_fwd_flag`, which is only known after the trip has been completed. This creates a major data leakage problem—one that would render the API unusable in a real-time setting, as users wouldn't have access to this value when requesting predictions.

After discovering this issue, two potential solutions were explored:

1. **Remove the feature and retrain the model:** This option would require rolling back several modeling decisions and retraining the pipeline—a time-consuming process.
2. **Introduce a secondary classifier:** A binary classification model could predict the flag using other available trip features. The predicted flag value would then be used as an input to the main model, allowing inference to proceed without depending on unavailable data.

This experience underscored the importance of carefully reviewing all features used in modeling and ensuring that each one is realistically available at prediction time. It also highlighted that some forms of data leakage may go unnoticed until late in the project lifecycle, particularly during deployment.

Final Reflections

- Building with a pipeline simplifies API and CLI integration.
- Data leakage can hide in subtle ways—especially with features only available post-event.
- Always validate that user input required during inference is both realistic and obtainable.
- Consider developing helper models to predict or approximate unavailable features if necessary.

5 Final Notes & Learnings

Throughout this project, I gained valuable hands-on experience across every phase of end-to-end machine learning development. From initial exploration to deployment, each stage came with its own challenges, learnings, and insights that will shape how I approach future projects.

To summarize, here are some of the key lessons reinforced during this journey:

- **Exploratory Data Analysis (EDA) is foundational:** A thorough EDA helped uncover meaningful patterns and relationships that directly guided feature engineering decisions. Strong insights during this phase can translate into significantly better model performance.
- **Assumptions can be misleading:** Not all observations made during EDA hold up in practice. Some features that seemed unhelpful turned out to improve performance, while others with strong correlation had little impact. Hence, testing all features during modeling—even the ones that seem weak—can yield unexpected benefits.
- **Model experimentation matters:** Although this project focused on feature engineering with a fixed Ridge Regression model, it became clear that trying multiple model types (e.g., XGBoost, Neural Networks) could further boost performance. This highlights the importance of model selection and hyperparameter tuning in future work.
- **Pipelines simplify deployment:** Building a clean preprocessing and modeling pipeline greatly streamlined the transition from development to production. With all preprocessing steps embedded, deploying and reusing the model became far easier and less error-prone.
- **Be mindful about data leakage:** One of the most important lessons came during the deployment phase, where I discovered a hidden data leakage involving the `store_and_fwd_flag` feature.

This emphasized the need to ensure that all features used in production are realistically available at inference time. Detecting such issues late in the pipeline can make fixes complex and time-consuming.

In conclusion, this project not only helped me build technical skills but also taught me how critical planning, validation, and iteration are in real-world machine learning workflows. I look forward to applying these learnings in future projects—especially in areas like robust pipeline design, model testing strategies, and deploying models reliably in production environments.

6 References

Dataset:
The dataset used in this project is private and cannot be shared publicly.

Libraries and Tools:

- `scikit-learn` – Machine learning models and preprocessing.
- `pandas` – Data manipulation and cleaning.
- `numpy` – Numerical computations.
- `seaborn`, `matplotlib` – Visualization.
- `FastAPI`, `argparse` – Deployment of the API and CLI interface.

Development Tools:

- Anaconda – Environment and package management.
- Jupyter Notebooks – Interactive development and testing.
- VS Code – Code editing and debugging.
- GitHub – Version control and collaboration.
- Postman – API endpoint testing.

External Resources:

1 "January 2016 United States blizzard," Wikipedia. Wikipedia Article
2 "JFK Airport Map." Google Maps
3 "LaGuardia Airport Map." Google Maps

Project GitHub Repository:

- github.com/b-4her/nyc-taxi-trip-duration-api – All the notebooks mentioned in this report, including the EDA and inference preparation notebooks, are available in the `notebooks/` directory of the repository.