INNOPOLIS UNIVERSITY

Final Report

New York City yellow taxi ride price prediction

Bulat Akhmatov Alexandra Vabnits Ruslan Izmailov

Nursultan Abdullaev

Abstract
Big data course

2025

1. Introduction

The goal of this project is to build a machine learning model that can accurately predict the total fare amount for yellow taxi rides in New York City. Being able to estimate ride fares ahead of time has several practical uses — from improving user experience in taxi apps to helping fleet operators and city planners understand pricing dynamics. Our model aims to make predictions based on features that are either known at the time of booking or collected during the ride, such as pickup and drop-off locations, trip distance, time of day, number of passengers, and payment method. By leveraging these variables, we hope to capture the key factors that influence fare prices and make the predictions as realistic and reliable as possible.

2. Data Description

The dataset we use comes from a public Kaggle source and contains detailed records of yellow taxi rides in NYC. It provides a solid foundation for exploring both the patterns in ride pricing and the effectiveness of different modeling approaches.

For the project, we used the yellow_tripdata_2016-03.csv snapshot, which contains records of yellow taxi rides in New York City for March 2016. The dataset comes from the NYC Taxi and Limousine Commission (TLC), which maintains and publicly shares trip data for various types of vehicles operating in the city. We specifically focused on Yellow Taxis, the iconic medallion cabs that are hailed directly from the street and operate only under strict regulation and a limited number of licenses.

This dataset is part of a larger initiative to provide transparency and support research around urban mobility and transportation infrastructure. At 1.91 GB in size, the March 2016 snapshot offers a rich sample of real-world taxi activity, capturing 12 210 952 rides with a variety of metadata for each trip.

The dataset includes a wide range of attributes, from timestamps and locations to fare breakdowns and payment types. Some key fields are:

- Pickup and drop-off timestamps: Indicate the start and end time of each trip.
- Geospatial data: Latitude and longitude for both pickup and drop-off points.
- Trip distance: Recorded by the taxi's meter.
- Passenger count: Input by the driver at the start of the ride.
- Fare components: Including base fare, taxes, surcharges, tips, tolls, and the total charged amount.
- Payment type: Captures whether the fare was paid by credit card, cash, or otherwise.
- Vendor and rate code info: Useful for understanding who recorded the data and under what fare conditions.

This mix of temporal, spatial, and financial data gives us a solid base for building a predictive model. It allows us to explore how different variables — like distance, time of day, or location — influence the final price of a ride.

3. Architecture of the Data Pipeline

Our project follows a multi-stage data pipeline that handles everything from data collection to model training and prediction. The goal is to create a robust, scalable flow that transforms raw CSV data into cleaned, structured input for machine learning models, and eventually outputs predictions and evaluation metrics. The pipeline architecture:

- Stage 1: Data Collection and Ingestion:
  - Input: Raw yellow taxi trip data (CSV) from Kaggle.
  - Output: Cleaned and structured Parquet files stored in HDFS.

  The first stage begins with downloading the dataset using the Kaggle API and unzipping the CSV file. We then store this data in a PostgreSQL database using a Python script that:
  - Creates necessary tables.
  - Imports the CSV data into the database.
  - Runs simple test queries to verify the import.

  Once the data is in PostgreSQL, we use Sqoop to transfer the tables into HDFS in Parquet format with Snappy compression. We also run a small Spark script to fix timestamp inconsistencies, which occur due to formatting issues during import.
- Stage 2: Data Storage, Preparation, and Exploratory Data Analysis (Hive Table Creation and Querying)
  - Input: Parquet files stored in HDFS.
  - Output: Partitioned Hive tables and query outputs (CSV).

  In the second stage, we use Hive to define a schema over the Parquet data and enable SQL-like querying.

  Our script (`stage2.sh`) connects to Hive using Beeline and:
  - Creates Hive tables with appropriate partitioning (`db.hql`)
  - Runs a series of analytical queries (`q1.hql` to `q7.hql`) to extract meaningful insights, like average fares, trip frequencies, and distribution patterns.

  The output of each query is saved as a CSV file for easy inspection and potential feature engineering.
- Stage 3: Predictive Data Analytics (Machine Learning Pipeline)
  - Input: Structured Hive data transformed and split by Spark.
  - Output: Trained models, predictions, and evaluation metrics.

  In the final stage, we launch a Spark-based ML pipeline (`ml_pipeline.py`). It performs the following:
  - Prepares training and testing datasets from the Hive tables.
  - Trains three baseline models and three tuned models.
  - Stores the trained models in HDFS.
  - Outputs predictions for each model to CSV files.
  - Writes an evaluation summary to a final CSV.

  This stage ties everything together by taking cleaned, queryable data and feeding it into a machine learning pipeline that can be evaluated and improved iteratively.

4. Data Preparation

This section outlines how we structured, loaded, and prepared the taxi trip data for analysis and machine learning. It covers three main parts: the database schema (via ER diagram), some sample data records, and the Hive setup we used to enable fast querying and partitioning.

4.1. Entity-Relationship (ER) Diagram

To represent the structure of our main table (`taxi_trips`), we created a simple ER diagram. While this project uses a single flat table rather than a multi-table relational model, the diagram still helps illustrate the key fields, their types, and constraints.

taxi_trips
VendorID (INT)
tpep_pickup_datetime (TIMESTAMP)
tpep_dropoff_datetime (TIMESTAMP)
passenger_count (INT)
trip_distance (FLOAT)
pickup_longitude (FLOAT)
pickup_latitude (FLOAT)
RatecodeID (INT)
store_and_fwd_flag (CHAR)
dropoff_longitude (FLOAT)
dropoff_latitude (FLOAT)
payment_type (INT)
fare_amount (FLOAT)
extra (FLOAT)
mta_tax (FLOAT)
tip_amount (FLOAT)
tolls_amount (FLOAT)
improvement_surcharge (FLOAT)
total_amount (FLOAT)

There are no foreign keys or separate dimension tables — all the data is contained in a single table. Constraints like valid payment_type ranges are implemented at the SQL level but not via relations.

4.2. Sample Data from the Dataset

Here are a few example records to illustrate the structure and values we worked with:

VendorID	Pickup Time	Dropoff Time	Passengers	Distance (mi)
2	“2016-03-01 00:00:00”	“2016-03-01 00:07:55”	1	2.50
1	“2016-03-01 00:00:00”	“2016-03-01 00:11:06”	1	2.90

Pickup (lon, lat)	Dropoff lon	RateCodeID	Flag	Payment Type
"−73.9767, 40.7652"	−74.0043	1	"N"	1
"−73.9835, 40.7679"	−74.0059	1	"N"	1

These records show typical taxi ride entries: start and end times, location coordinates, distance, and fare-related information.

4.3. Hive Table Creation and Data Transformation

Once the data was ingested and stored in PostgreSQL, we moved it through several transformation steps to prepare it for analysis and machine learning. Here's a breakdown of what happens:

- In Stage 1:
  - We downloaded the raw CSV dataset from Kaggle using a shell script.
  - Then we loaded the data into a PostgreSQL table (`taxi_trips`) via a Python script that reads from create_tables.sql and import_data.sql.
  - Using Sqoop, we transferred the data from PostgreSQL into HDFS as compressed Parquet files, a columnar format that's efficient for big data processing.
  - A Spark job (`fix_parquet_timestamps.py`) was then executed to fix formatting issues in the timestamp columns.
- In Stage 2:
  We created two Hive external tables:
  - `taxi_trips`: a base table directly reading from the fixed Parquet files.
  - `taxi_trips_part`: a partitioned table (partitioned by payment_type) for improved query performance. The decision was to partition data directly by the `payment_type` column due to its low cardinality and other attributes, which made it the most suitable column for partitioning.

  Hive dynamic partitioning was enabled and used to load data from the base table into the partitioned table. Several HiveQL scripts were then executed to run queries and generate summary CSV files.

  We chose not to use bucketing in this project because our queries did not involve frequent joins or large-scale sampling where bucketing would significantly help. Partitioning by `payment_type` alone was sufficient to improve performance for the types of queries we were running.

5. Data Analysis

To better understand which features influence the fare prediction, we started by calculating the correlation between each input variable and the total fare amount. This gave us an initial sense of which features might be the most informative for our model.

As expected, the strongest correlation by far is with fare_amount, which makes sense since total fare is essentially the base fare plus extras like tips, tolls, and taxes. Interestingly, trip_distance also shows a moderate positive correlation with the total fare (0.4172), confirming the intuitive idea that longer trips tend to cost more.

Most of the additional charges — like extra, tip_amount, and tolls_amount — have small but noticeable positive correlations, suggesting they do contribute to the overall price, but not as significantly as distance or the base fare. Fields like pickup_longitude and latitude show zero correlation, likely because geographic coordinates on their own don't convey much without being transformed into zones or distances.

Passenger_count, mta_tax, and improvement_surcharge all have very low or near-zero correlation with total fare, meaning they are likely less useful as direct predictors in their raw form.

Metric	Total (Sum)
corr_total_amount_passenger_count	265µ
corr_total_amount_trip_distance	0.4172
corr_total_amount_pickup_longitude	0
corr_total_amount_pickup_latitude	0
corr_total_amount_fare_amount	0.9996
corr_total_amount_extra	0.0631
corr_total_amount_mta_tax	-0.0182
corr_total_amount_tip_amount	0.0692
corr_total_amount_tolls_amount	0.0593
corr_total_amount_improvement_surcharge	-0.0046

Figure 2: Correlation Analysis

To dig deeper into fare behavior, we explored how payment type affects tipping, since tips are a meaningful part of total fare and often vary depending on how a passenger pays.

The pie chart below shows the overall distribution of payment methods in our dataset. Most rides were paid using a credit card (payment_type = 1), followed by cash payments (payment_type = 2). The other types — "no charge" and "dispute" — are rare in comparison.

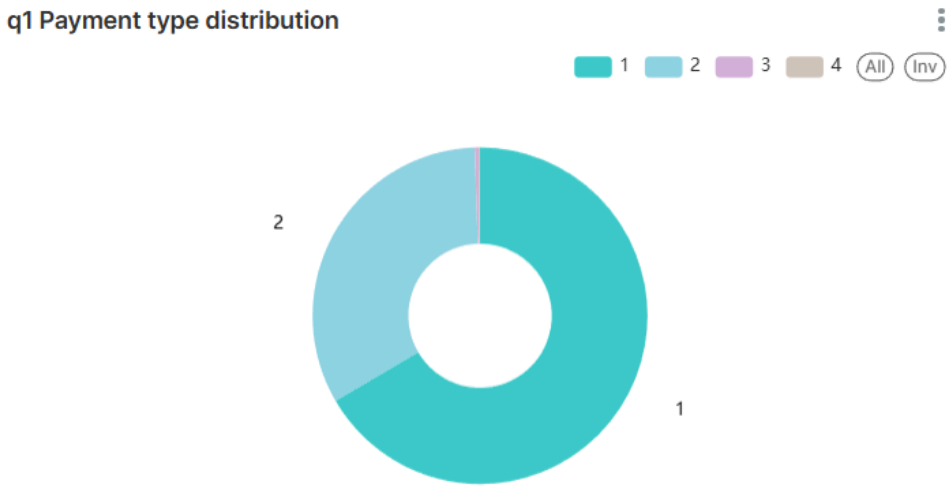

Figure 3: Payment Type Distribution

Next, we looked at tip statistics by payment type. Since tips are only recorded for credit card transactions (cash tips aren't captured), it makes sense that only payment_type = 1 shows non-zero values for average and median tips. The credit card group also contains some extremely high maximum tips,

which could reflect outliers or data entry errors. Other payment types have negligible or no tipping data.

Figure 4: Tip Statistics by Payment Type (Log Scale)

We also looked at how fares vary by the number of passengers. The bar chart below shows average total fare, base fare, and tip amounts for different passenger counts.

At first glance, the chart suggests a strong increase in fare amounts for higher passenger counts, especially for 7 to 9 passengers. However, this pattern doesn't reflect in the correlation we saw earlier — passenger count had near-zero correlation with total fare. This apparent contradiction is mainly because high passenger counts are extremely rare in the dataset. So even though their averages are high, they don't influence the overall trend much due to low sample size.

On the other end, passenger count of 0 — which likely represents data entry errors or non-passenger trips — also has a surprisingly high average fare, but again, it's uncommon.

For the majority of rides (with 1 to 6 passengers), the differences in average fare are minimal, which helps explain why the overall correlation is weak. The variation isn't strong enough across the bulk of the data to matter statistically.

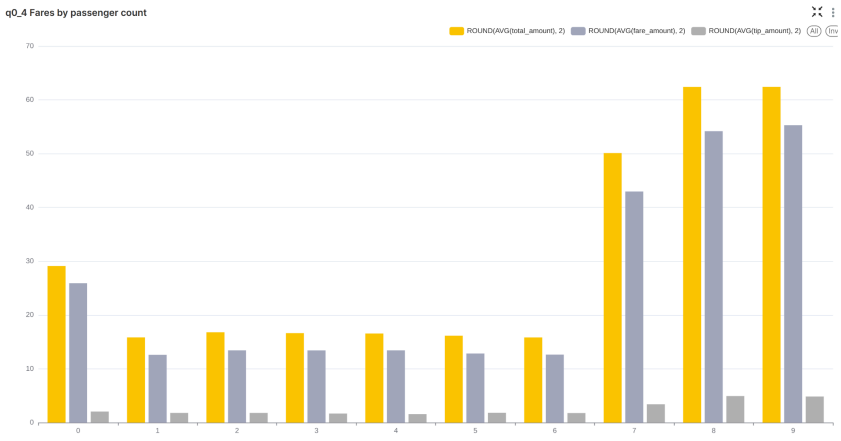

Figure 5: Average Fares by Passenger Count

To better understand why passenger count has such a weak correlation with total fare, we looked at how ride frequency is distributed across different passenger counts. As the pie chart shows, the vast

majority of rides involve just one passenger, followed by two-passenger trips. Together, they make up the overwhelming bulk of the data.

Counts above 2 — especially 4 through 9 — are relatively uncommon, and some (like 7, 8, or 9) are extremely rare. Even though these higher passenger counts showed noticeably higher average fares in the previous bar chart, they represent such a small portion of the data that their influence on overall correlation is minimal.

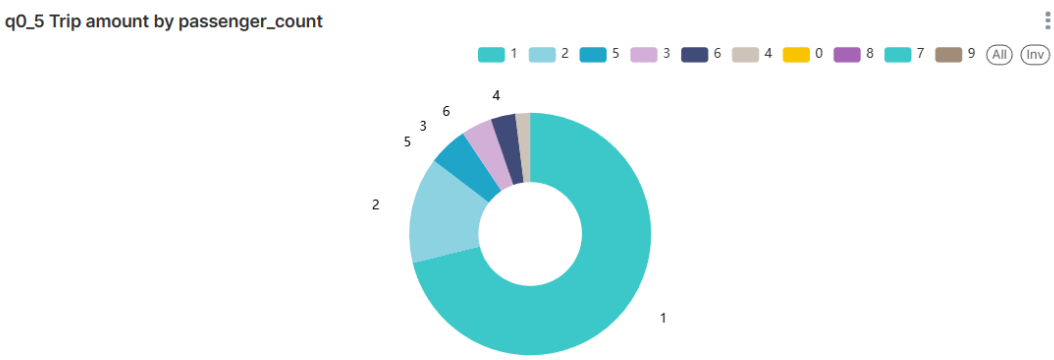

Figure 6: Trip Count by Passenger Number

To enrich the dataset, we created a new feature: trip duration, calculated as the difference between drop-off and pickup timestamps. This gave us a better sense of how long rides actually lasted, which is often more informative than distance alone — especially in a city like New York where traffic can be unpredictable.

The histogram below shows the distribution of trip durations (in seconds). Most trips are fairly short, clustering around the 5 to 10-minute range (roughly 300 to 600 seconds). After that, the frequency steadily drops as duration increases, forming a long right tail.

There are a few extremely long trips in the data, but they're rare. These outliers may reflect airport rides, intercity trips, or even anomalies.

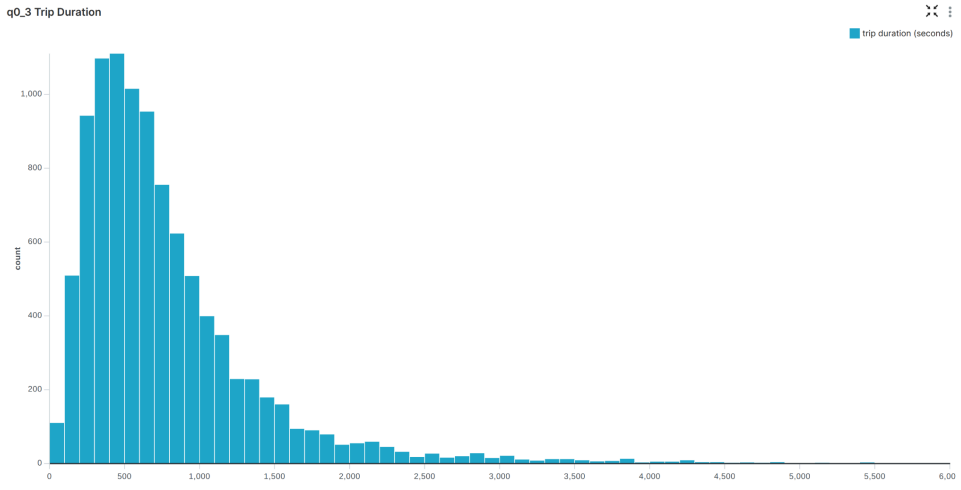

Figure 7: Distribution of Trip Duration

Distance is one of the most important predictors for fare amount, so we analyzed how both fare and frequency change across different distance ranges.

The chart below shows average total and base fares grouped by trip distance (in miles). As expected, there's a clear upward trend: the longer the trip, the more it costs. The increase is relatively smooth and consistent, especially between the 0–1, 1–3, and 3–5 mile ranges. For trips above 10 miles, the average total cost spikes, reaching over $50 — likely reflecting airport transfers or trips across boroughs.

The gap between total and base fare also widens with distance, suggesting that longer rides accumulate more tips, tolls, and surcharges. This supports the idea that fare extras are distance-sensitive, even if not directly proportional.

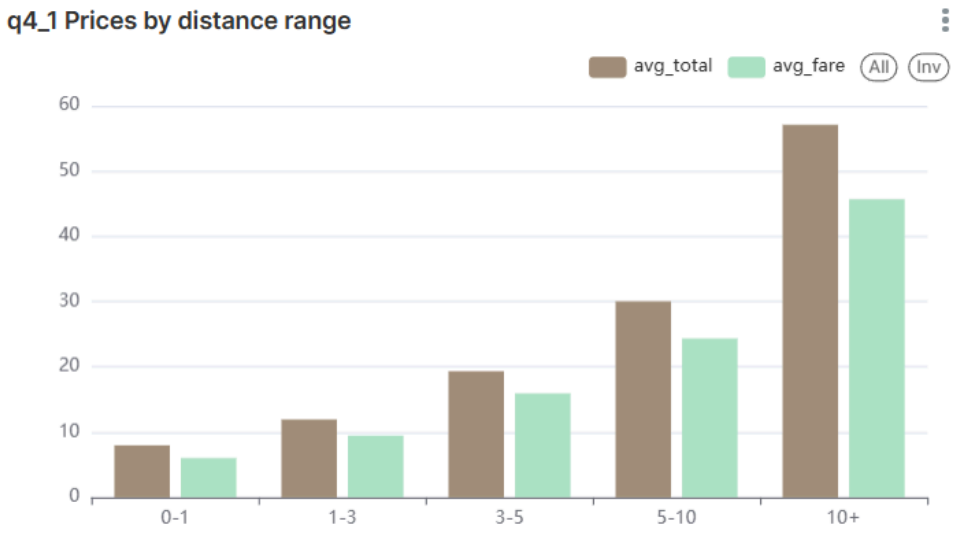

Figure 8: Average Fare and Total Cost by Distance Range

The chart below highlights how frequent different trip distances are. Most rides fall in the 1–3 mile range, followed by very short trips (under 1 mile). Longer trips, especially those over 10 miles, are less common. This distribution is important because it reminds us that while long-distance trips generate higher fares, they represent a smaller share of the data.

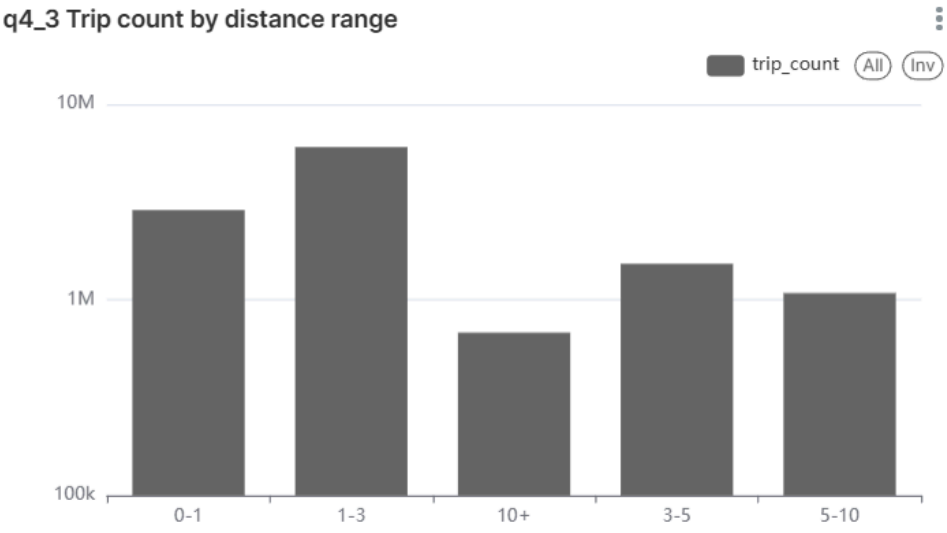

Figure 9: Trip Count by Distance Range

Before training any machine learning models, we reviewed basic descriptive statistics for key numerical features related to fare pricing. This included the fare itself, surcharges, taxes, tips, and total amount.

One issue became immediately clear: several features contain unnatural negative values, such as a minimum fare of –$68 or an mta_tax of –$0.50. These values are clearly invalid — fares can't logically be negative — and likely stem from input errors, system glitches, or special refund cases incorrectly recorded.

Ignoring these would bias the model or cause instability during training, so we marked these values for cleanup in the preprocessing phase. In addition, we observed that:

- The mean total fare is around $16, with a median of $11.80, which confirms a right-skewed distribution — also seen in earlier charts.
- The maximum tip and toll amounts are quite high, suggesting the presence of extreme outliers.

stat	fare_amount	extra	mta_tax	tip_amount	tolls_amount	improvement_surcharge	total_amount
count	123k	123k	123k	123k	123k	123k	123k
median	9.5	0	0.5	1.35	0	0.3	11.8
max	400	4.5	0.65	120.05	100.99	0.3	520.35
mean	12.76	0.3445	0.4976	1.79	0.3161	0.2997	16
min	-68	-1	-0.5	0	0	-0.3	-68.8
std	11.04	0.5087	0.0377	2.49	1.42	0.0124	13.63

Figure 10: Summary Statistics for Fare-Related Features

Overall, the dataset appears quite natural, with trends and correlations that generally make sense. Fare amounts increase with trip distance, tips are tied closely to credit card payments, and longer trips naturally cost more — both in base fare and extras. While most patterns are intuitive, there are occasional outliers that don't make sense, like negative fares or unusually high tips, which we flagged for cleanup. Interestingly, geographic coordinates (pickup and dropoff locations) show almost no direct linear correlation with total fare. This suggests that raw latitude and longitude aren't very informative on their own and may require transformation (like calculating zones or distance) to be useful. On the other hand, trip distance stood out as one of the most important and reliable predictors of fare, which aligns well with how taxi pricing typically works.

6. Machine Learning Modeling

Goal: Fare Prediction (Regression Problem)

The goal of our modeling stage was to accurately predict the total fare amount for NYC taxi rides. This is a regression task, since we're estimating a continuous numerical value.

To tackle this, we built a full Spark-based ML pipeline that includes feature extraction, training, tuning, and evaluation — all automated and scalable.

6.1. Feature Extraction and Preprocessing

We started by reading the cleaned trip data from Hive tables. From there, we applied a series of custom transformations to engineer meaningful features:

- Converted timestamps into usable formats.
- Extracted temporal features like pickup/dropoff hour and month.
- Applied cyclical encoding to capture time-of-day and seasonal effects.
- Selected relevant features like coordinates, trip distance, and passenger count.
- Used VectorAssembler and StandardScaler to build normalized input vectors for modeling.

This preprocessing was wrapped into a reusable pipeline and applied to both the training and testing splits (80/20), resulting in around 9 million training samples and 2 million test samples.

6.2. Model Training and Tuning

We trained three different regression models:

1. Random Forest
2. Gradient Boosted Trees
3. Decision Tree

Each model was first trained with default hyperparameters to establish a baseline. Then, we performed grid search tuning with cross-validation on the two most promising models: Random Forest and Gradient Boosting.

The training process was designed to be fully automated — models, predictions, and evaluation metrics were all saved to HDFS and local storage for later inspection.

6.3. Evaluation Results

Performance was evaluated using RMSE (Root Mean Squared Error) and R^2 (coefficient of determination):

Model	RMSE	RMSE (tuned)	R^2	R^2 (tuned)	Inference Time (ms)
Gradient Boosting	5.43	5.20	0.83	0.85	0.04
Random Forest	6.23	5.35	0.79	0.84	0.05

Tuning clearly improved both models, with Gradient Boosting slightly outperforming Random Forest in both accuracy and speed. Given the large dataset and real-world use case, these results suggest the models generalize well while maintaining fast inference.

7. Data Presentation

To make our project results more interactive and accessible, we built a web dashboard using Apache Superset, populated from Hive tables created in Stage 4 of the pipeline. These tables (mq1, mq_model_1, and mq_model_2) are used to feed insights, predictions, and model performance data directly into the dashboard. This setup allows us to tell the full story of the dataset — from raw input to predictive modeling — through visual exploration.

7.1. Dashboard Description

The dashboard is organized into three main tabs:

1. Data Description: Gives an overview of the dataset and schema, including types of features (categorical, continuous, etc.).
2. Data Insights: Focuses on exploratory data analysis (EDA), helping us understand trends, distributions, and feature importance.
3. Price Regression: Displays model performance metrics, error analysis, and prediction comparisons.

7.2. Chart Descriptions and Findings

- Figure 1 – q0_1: Average Total Amount by Day

This time-series line chart tracks the average total fare per day. While total amount doesn't depend heavily on the date, we observed a clear day-of-week pattern — small dips repeat regularly, suggesting quieter days (likely weekends or mid-week lows).

- Figure 2 – q0_2: Trip Duration by Vendor

A box plot comparing trip durations by vendor ID. Both vendors (1 and 2) show fairly similar duration distributions, though vendor 1 has slightly more variation. No major anomalies were seen here, indicating that vendor is likely not a strong standalone predictor.

- Figure 3 – q0_3: Trip Duration Histogram

This histogram reveals that most trips are short — typically under 15 minutes — and that trip durations follow a long-tailed distribution. A small number of outlier trips last far longer, potentially due to traffic, distance, or anomalies.

Key Takeaways:
1. Day-of-week patterns exist in fare data, even if total amount isn't time-dependent overall.
2. Vendors behave similarly, suggesting their influence on fare may be limited.
3. rip duration and distance are strong and meaningful features, consistently shaping fare amounts.
4. The dashboard also serves as a centralized way to explore prediction results and compare models visually, making it easier to iterate and communicate findings.

8. Conclusion

In this project, we built a full data pipeline to predict taxi fare amounts in New York City using real-world ride data. We collected, cleaned, and transformed the data, explored it through visual analysis, and trained multiple machine learning models to solve a regression problem. Among the models tested, Gradient Boosting performed best after tuning, achieving strong accuracy and fast inference. The entire workflow — from raw CSVs to model evaluation and dashboard presentation — was automated and deployed using big data tools like Hive, Spark, and Superset. Overall, the project demonstrated how structured pipelines and thoughtful feature engineering can lead to interpretable and effective ML solutions at scale.

9. Reflections

9.1. Challenges and Difficulties

Like any real-world project, we faced several practical challenges along the way:
1. Cluster uptime and stability: Occasionally, the cluster would be down or slow to respond, which made debugging and testing less predictable.
2. Training time: Grid search tuning for some models took up to 4 hours per run, which limited how many parameter combinations we could try within the project timeline.
3. Collaboration environment: Working in a shared Jupyter session with no Git extension made version control tricky and increased the risk of overwriting or conflicting edits between teammates.

9.2. Recommendations

If we were to repeat or extend this project, we'd recommend:
- Improved resource scheduling, such as reserving cluster time in advance or setting up alerts for downtime.
- Smaller or smarter parameter grids for tuning, possibly using techniques like Bayesian optimization to cut down training time.

- A more refined Git-backed workflow, even if working in Jupyter, to improve collaboration and reduce merge conflicts. Using branches and regular commits would have made team contributions more organized and traceable.

9.3. Team Contributions

TO DO