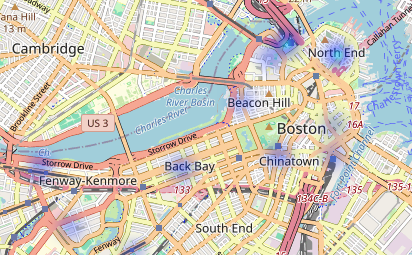
**Rideshare Pricing and Demand Dynamics**

***A Machine Learning Exploration Using***

***Uber and Lyft Data in Boston***



Authors: Kezuwera Dibeley, Shreeya Sampat, Victor MacEdo

OMSBA 5500: Capstone Project in Business Analytics

Professor Ben Kim

June 2025

**Table of Contents**

1. **Abstract / Executive Summary**………………………………………………………..3
2. **Introduction**…………………………………………………………………………….4  
     2.1 Background  
     2.2 Problem Statement  
     2.3 Research Questions
3. **Data Description and Sources**…………………………………………………………5  
     3.1 Primary Rideshare Dataset  
     3.2 Income Data Integration  
     3.3 ZIP Code Mapping  
     3.4 Final Dataset Overview
4. **Literature Review**………………………………………………………………………7  
     4.1 Review of Article 1  
     4.2 Review of Article 2  
     4.3 Review of Article 3  
     4.4 Review of Article 4  
     4.5 Review of Article 5  
     4.6 Review of Article 6
5. **Data Pre-Processing**……………………………………………………………………10  
     5.1 Data Integration  
     5.2 Data Cleaning and Missing Value Handling  
     5.3 Feature Engineering  
     5.4 Handling Class Imbalance  
     5.5 Scaling and Dimensionality Reduction
6. Modeling and Evaluation………………………………………………………………12  
     6.1 Surge Pricing Classification  
     6.2 Weather-Based Price Regression  
     **6.3 Income-Based Pricing and Surge Analysis**  
     6.4 Demand Forecasting with Time Series  
     6.5 Top Performing Models and Metrics
7. **Discussion**……………………………………………………………………………….21  
     7.1 Domain Knowledge Insights  
     7.2 Methodological Contributions  
     7.3 Pricing Fairness Implications
8. **Conclusion**……………………………………………………………………………....23  
     8.1 Summary of Findings  
     8.2 Limitations  
     8.3 Recommendations for Future Work
9. **References**……………………………………………………………………………….26
10. **Appendices**………………………………………………………………………….….A-1  
      A.1 Code with Comments  
      A.2 Code Snippets with Outputs   
      A.3 Supplementary Visualizations and Tables
11. **Abstract / Executive Summary**

Rideshare platforms such as Uber and Lyft have significantly reshaped urban mobility by offering flexible, on-demand transportation through real-time pricing algorithms. These platforms rely on dynamic pricing models, often called “surge pricing,” to balance supply and demand. While this system enables greater operational efficiency and responsiveness, the actual factors influencing surge pricing and demand behavior remain complex and largely opaque to both users and policymakers. This project investigates these underlying dynamics using real-world rideshare data from the Boston metropolitan area. Our analysis draws from a dataset of over 690,000 ride records collected over a three-week period in late 2018. To enrich the dataset and add depth to our analysis, we merged ZIP code-level income data and geographic mappings, allowing us to incorporate socioeconomic and spatial context into our modeling.

We focused on three core research questions. First, what features are most strongly associated with surge pricing? Second, how do weather conditions affect ride prices? And lastly, can rideshare demand be accurately forecasted using variables like time, location, and weather? To answer these questions, we implemented a structured data science pipeline that involved data cleaning, merging, feature engineering, and exploratory visualization. Key features included ride timing (hour, day type, peak hours), weather indicators (temperature, humidity, precipitation), geographic factors (ZIP code, income brackets), and ride-specific attributes (distance, cab type).

For the surge classification task, we trained and evaluated a variety of machine learning models including Decision Tree, Random Forest, Gradient Boosting, and Neural Network classifiers. LinearSVC with calibration was also included to improve probability predictions. Model performance was evaluated using K Fold cross-validation and tested on a 20% holdout set. Key metrics included accuracy, F1 score, and area under the ROC curve (AUC). The Random Forest classifier emerged as the best-performing model, achieving an AUC of 0.92. Important predictors included ride distance, hour of the day, temperature, and ZIP-level income. These results revealed strong temporal and spatial signals in surge events.

To analyze weather-based price variation, we applied regression techniques using a log-transformed version of the ride price as the target variable. The regression models included Linear Regression, Random Forest Regressor, and Gradient Boosting Regressor. Evaluation metrics included R², Mean Absolute Error (MAE), and Root Mean Squared Error (RMSE). Although the overall R² values were modest, the models confirmed that colder, windier, and rainier conditions were generally associated with higher ride prices, suggesting that inclement weather likely drives up demand or limits supply.

We also explored demand trends using time series modeling. Hourly ride counts were used to train both ARIMA and Prophet models, with Prophet selected as the final model due to its better interpretability and ability to handle seasonality and trend shifts. The model successfully captured daily and weekly ride patterns, with clear peaks during weekday commute hours and dips around weekends and holidays. The final forecast achieved a Mean Absolute Percentage Error (MAPE) of 12.5%, indicating strong predictive capability over the short term.

A critical part of our analysis was understanding how income levels intersect with pricing and demand. Our income-based segmentation revealed that higher-income ZIP codes not only experienced higher average fares, but also more frequent surge pricing events. This pattern raised important questions about pricing equity, as it suggests potential socioeconomic bias in algorithmic pricing, even if unintentional. While this may reflect demand density or traffic patterns, it highlights the importance of considering fairness when designing or auditing dynamic pricing models.

This project demonstrates how combining machine learning with thoughtful feature engineering and domain knowledge can uncover valuable insights into rideshare operations. The results have practical implications for both companies and users, supporting improved fare prediction, better demand planning, and more equitable pricing strategies. As the rideshare industry continues to evolve, especially with the integration of autonomous vehicles and real-time predictive pricing, further research will be needed to ensure these systems are not only efficient but also fair and transparent. Future work may explore the role of live traffic, event-based demand shocks, or rider demographic data, while also emphasizing ethical practices in data collection and model interpretation.

1. **Introduction**

***2.1 Background***

The rise of rideshare platforms such as Uber and Lyft has transformed urban mobility by offering greater flexibility and convenience to commuters. These services use dynamic pricing, a system that adjusts fares in real time based on demand, supply, location, and external factors such as weather or traffic. While dynamic pricing helps balance supply and demand, the mechanisms behind these fare changes, especially surge pricing, are often unclear to both riders and policymakers.

Our exploration of rideshare activity in Boston revealed that pricing patterns fluctuate significantly based on the time of day, day of the week, weather conditions, and neighborhood characteristics. Weekday mornings between seven and nine, and evenings between four and seven, particularly in areas like Back Bay and the Financial District, consistently show high demand that often triggers surge pricing. Weather also plays an important role. Rain and colder temperatures tend to increase ride requests, with surge pricing occurring nearly twice as often on rainy days. During periods of extreme weather or major public events, prices may rise to nearly three times the normal fare. These unpredictable spikes can lead to rider dissatisfaction and raise questions about pricing fairness.

Machine learning provides an opportunity to better understand these fluctuations and offer more transparency. By identifying patterns across large and complex datasets, machine learning models can help predict when and where surge pricing is likely to occur. This supports better driver distribution, improves service reliability, and helps customers make informed decisions. Our project uses machine learning not only to model pricing behavior, but also to understand how external conditions and socioeconomic factors influence rideshare systems in a major city.

***2.2 Problem Statement***

Although surge pricing is a widely used feature in rideshare platforms, its inner workings are not well understood by the public. Riders often encounter increased fares without clarity on what caused them. From the perspective of service providers, this lack of transparency can reduce customer trust. From a policy and equity standpoint, it raises important concerns, especially if certain neighborhoods are affected more than others.

This project seeks to examine the underlying factors contributing to surge pricing, explore how weather affects fare prices, and determine whether demand patterns can be predicted using temporal, spatial, and environmental data. By integrating trip-level rideshare data with weather and income datasets, we aim to uncover insights that can improve pricing transparency and operational planning in urban transportation.

* 1. **Research Questions**

This study is guided by three core questions:

* 1. What factors are most associated with surge pricing?

We apply classification models such as decision tree, random forest, gradient boosting, and neural network to understand the timing and location of surge pricing events.

* 1. How do weather patterns influence rideshare pricing?

We use regression analysis with variables such as rain, temperature, and wind to measure their impact on fare prices.

* 1. Can rideshare demand be forecasted using time, location, and weather data?

We apply time series forecasting models to predict future demand trends and support better planning by rideshare companies.

1. **Data Description and Sources**

***3.1 Primary Rideshare Dataset***

The primary dataset used in this study is titled *Uber and Lyft Dataset Boston, MA*, sourced from Kaggle. It contains over 690,000 individual trip records collected in Boston between late November and mid-December 2018. Each record represents a single ride request and includes detailed trip-level information such as cab type, pickup and drop-off neighborhoods, timestamp, distance, fare amount, and surge multiplier. Additionally, the dataset includes weather variables such as temperature, precipitation, humidity, and wind speed for the time and location of each ride.

Both Uber and Lyft data are included, allowing for platform-level comparisons. Although the dataset covers only a three-week window, it is dense and diverse enough to reveal meaningful patterns in pricing, demand fluctuations, and temporal trends. This dataset served as the primary input for our classification, regression, and time series models.

***3.2 Income Data Integration***

To provide socioeconomic context, we integrated a second dataset, *us\_income\_zipcode.csv*, which contains median household income data by ZIP code across the United States, sourced from the U.S. Census and available through Kaggle. By linking each trip’s pickup and drop-off locations to their respective ZIP codes, we were able to assign income values to both ends of each ride. This allowed us to investigate how pricing behaviors, including surge pricing, vary across different income levels.

This additional layer of information enabled us to examine important equity-related questions, such as whether surge pricing occurs more frequently in higher-income neighborhoods and whether certain areas are more affected by price fluctuations.

***3.3 ZIP Code Mapping***

Since the rideshare dataset used neighborhood names while the income data was organized by ZIP codes, we used a third dataset to bridge the gap. The *ZIP\_Locale\_Detail.xls* file, published by the United States Postal Service, maps neighborhood names to their corresponding ZIP codes. This was essential for accurately merging the income data with the rideshare data.

We performed data cleaning to standardize neighborhood names and used these mappings to assign ZIP codes to each ride’s pickup and drop-off locations. This step enabled precise spatial analysis and allowed for the aggregation of data by ZIP code to uncover geographic trends in pricing and demand.

***3.4 Final Dataset Overview***

After extensive cleaning, transformation, and merging of the three datasets, we created a unified and structured dataset with over 690,000 records. Key preprocessing steps included:

* Removing duplicate entries and dropping irrelevant columns such as ride IDs and unused weather variables
* Handling missing values, especially in the price column, where approximately eight percent of records were incomplete
* Standardizing categorical variables such as cab type and neighborhood names
* Normalizing continuous variables including fare, distance, and temperature
* Engineering new features such as surge classification, hour of day, day type (weekday or weekend), and ZIP-code-level income

The final dataset included a comprehensive set of features across ride behavior, time, weather conditions, and socioeconomic context.

A summary of key variables is provided in Table 3.1.

|  |  |  |
| --- | --- | --- |
| **VARIABLE** | **DESCRIPTION** | **EXAMPLE** |
| price | Ride fare in USD | Median: $10.50 (Range: $2–$80) |
| surge\_multiplier | Surge pricing factor | 1.0 to 3.5 |
| distance | Trip distance in miles | Median: 2.5 (Range: 0.1–10.0) |
| hour, day, month | Temporal features | 0–23, 1–31, 11–12 |
| cab\_type, ride\_type | Platform and service level | UberX, Lyft Plus, etc |
| temperature, precipitation | Weather at time of trip | 20–60°F, 0 to 1.2 inches/hour |
| median\_income\_zip | Household income for pickup ZIP code | $33,000 to $103,000 |

**Table 3.1: Key Variables and Summary Statistics**

It is important to note that this dataset reflects rideshare activity limited to a three-week period in November and December of 2018 in the city of Boston. As a result, the findings may be influenced by seasonal patterns, holiday travel behavior, and specific local events that occurred during that time. These temporal constraints were considered when interpreting the results, and limitations in generalizability are discussed further in the conclusion.

In addition, while this study successfully combined three independent data sources, rideshare trip data, ZIP-code-level income data, and ZIP-to-location mapping files, the integration process presented challenges. Some records were excluded due to missing or inconsistent location identifiers, and slight variations in neighborhood naming may have affected geographic accuracy. Despite these constraints, the merged dataset maintained a high level of completeness and supported the development of robust classification, regression, and forecasting models.

1. **Literature Review**

***4.1 Review of Article 1***

Article 1: Ridesourcing, the Sharing Economy, and the Future of Cities

The research conducted by Jin et al. (2018) examines real-world effects of ride-hailing services Uber and Lyft to determine if they fulfill the marketing promises made by the companies. The research examines three essential matters including efficiency, equity, and sustainability. The research findings contradict industry statements by demonstrating that ride-hailing services typically create more traffic problems. The practice of vehicles driving without passengers between trips increases traffic congestion most notably in central business districts where they contend with mass transit systems instead of replacing personal vehicle use. Systemic unfairness becomes apparent through algorithmic price increases in low-income neighborhoods while drivers face unstable earnings without job protection. The environmental advantages of ride-hailing services are small because users maintain ownership of their vehicles thus ride-hailing services increase traffic congestion. The research reveals its most significant achievement through its analysis of false statements made by ride-hailing companies. The authors contend that the "sharing economy" label misleads people because authentic carpooling requires sharing rides with unknown passengers. The technology-based operation of UberX resembles traditional taxi services while imposing concealed expenses on urban areas. The study demonstrates that ride-hailing services promote greater vehicle emissions through increased driving behavior which bypasses regulations that govern conventional public transportation. The authors recommend that policymakers should ignore corporate marketing terms and establish policies through empirical data by implementing electric vehicle requirements for ride-hailing fleets and density controls for busy regions. The research establishes guidelines for cities to manage these services instead of letting companies take control.

***4.2 Review of Article 2***

Article 2: Modeling and Managing Mixed On-Demand Ride Services of Human-Driven and Autonomous Vehicles: A Monopoly Market Analysis

Mo et al. (2022) examines the optimal operation of ride-hailing platforms that integrate human-driven cars (HVs) with self-driving vehicles (AVs). The research shows that operating with a combination of autonomous vehicles and human-driven cars is more complicated than just integrating self-driving cars. The researchers created a mathematical model to evaluate various scenarios which led to three essential findings. The "wild goose chase effect" occurs when an excessive number of available cars creates traffic congestion because they drive on empty streets. The inefficiency of this process cancels out any potential advantages that additional vehicles might bring. AVs increase system efficiency through their ability to reduce traffic congestion which leads to higher earnings for human drivers. The pricing models which operate statically prove ineffective for mixed fleets because platforms need to adjust prices automatically according to real-time demand and road conditions to achieve supply and demand equilibrium. The research identifies the best possible combination of vehicles between autonomous and human-driven cars at sixty percent AVs and forty percent HVs. The specific vehicle distribution leads to maximum profit gains for the platform alongside maintaining driver compensation and rider affordability. Autonomous vehicles operate efficiently on regular busy routes, yet human drivers maintain flexibility for navigating complicated routes. The authors recommend that cities establish regulations for fleet size limits because an unlimited number of self-driving vehicles might create street congestion. The model presents a path forward for companies and policymakers who need to manage the shift to autonomous ride-hailing by showing that effective management of human and machine roles stands as crucial to success beyond technological advancements.

***4.3 Review of Article 3***

Article 3: Predicting Surge Pricing of Ride Sharing Services Using Weather and Temporal Data

This study focuses on understanding how surge pricing works in rideshare services like Uber and Lyft. The authors explore how external factors such as weather conditions, time of day, and traffic patterns can help predict when surge pricing is likely to happen. The goal is to build a model that can give early warnings or help users and companies better prepare for these price increases. To do this, the researchers collected rideshare data that included weather details like temperature and rainfall along with date and time information. They used machine learning models like Random Forest and Decision Trees to predict if a ride would have surge pricing. Among the models tested, Random Forest performed the best in terms of accuracy. The study is useful because surge pricing can be confusing or frustrating for customers, especially when it happens without warning. The paper shows that it's possible to forecast surge pricing by using data that's already being collected, such as ride time and weather. This could help rideshare platforms give better price estimates and help customers choose better times to ride.

***4.4 Review of Article 4***

Article 4: Impacts of Trip Characteristics and Weather Condition on Ride-Sourcing Network: Evidence from Uber and Lyft

This study explores how different trip features and weather conditions affect the way rideshare services like Uber and Lyft operate. The researchers focused on how factors like pick-up and drop-off locations, time of day, trip distance, and weather including rain, temperature, and wind influence three main things: pick-up wait time, trip duration, and fare. The goal was to understand how these conditions affect rider experience and system performance. To carry out the study, the authors collected ride data from Uber and Lyft developer platforms and combined it with weather information from Yahoo Weather API. The data came from trips taken in Philadelphia during the month of June 2018. They used different types of statistical analysis, such as regression models, to identify patterns between weather, trip details, and rideshare outcomes. The findings showed that weather had a big impact on the ride network. On weekdays, bad weather like rain or high winds led to longer wait times and more expensive rides, likely because more people were requesting rides while fewer drivers were available. But on weekends, the opposite happened; bad weather caused a drop in ride demand, so wait times and prices were lower. The study also found that trips closer to the city center were faster and had shorter wait times compared to those in suburban areas. This research is helpful because it shows that rideshare networks respond differently depending on the time of the week and outside conditions. These insights can help companies adjust how they assign drivers and set prices, and they can also be useful for city planners thinking about how to improve transportation access.

***4.5 Review of Article 5***

Article 5: A Comprehensive Overview of Deep Learning for Algorithmic Pricing in Ride-Sharing Platforms

Chirita and Chirita (2024) study the impact of deep learning technology on pricing algorithm development for ride-hailing services including Uber and Lyft. The research investigates how these sophisticated methods enhance basic pricing systems by handling extensive real-time data sets. The research reveals three essential advantages which include precise minute-by-minute demand predictions for price adjustments and driver-passenger matching for time reduction and individualized pricing fairness. The authors stress that these systems need close monitoring to avoid algorithmic bias. The most useful outcome demonstrates how neural networks address unique pricing problems. Convolutional Neural Networks (CNNs) examine location-based patterns to understand price increases at concert venues. Long Short-Term Memory (LSTM) networks demonstrate superior demand surge prediction capabilities through their ability to learn historical ride patterns across weeks. These tools enable platforms to manage their supply and demand better than human planners would have been able to do. The research stresses that businesses need to establish protection measures including transparency reports and bias audits to prevent powerful algorithms from creating unfair disadvantages for specific neighborhoods and rider groups. The research establishes guidelines for creating advanced pricing solutions with ethical standards that benefit both business operations and urban communities.

***4.6 Review of Article 6***

Article 6: A Novel Approach to Analyze Uber Data Using Machine Learning

The research conducted by Narasimha Rao et al. (2023) examines the machine learning algorithms Uber employs to enhance its pricing structure and driver operations. In this research, Linear Regression and Random Forest models were employed to predict ride demand and fare structures. The research demonstrates that Uber's data operates according to regular patterns because the implemented models successfully predict 83% of demand variations. The analysis reveals that ride demand increases during weekday rush hours and weekend late nights. The research concludes that successful outcomes depend heavily on data preprocessing which includes both value replacement and fare entry correction. The visualization of data helps professionals detect patterns including which areas experience the most activity at times. The research study presents useful applications that benefit both Uber platform users and its drivers. Random Forest models demonstrate better performance than basic models when predicting real-world fare estimates because they handle complex scenarios such as sudden rainstorms and unexpected demand spikes. The authors propose that Uber should implement real-time analysis tools that would enable instant reactions to unforeseen changes such as sports game endings before time or subway delays. These insights provide drivers with better methods to optimize their work by positioning themselves in business districts during evening commutes. The research proves that data-based strategies enhance operational efficiency for both ride-hailing companies and their drivers.

1. **Data Pre-Processing**

***5.1 Data Integration***

We began by merging three separate datasets: rideshare trip data, weather information, and ZIP code–level income data. Joins were performed using common fields such as timestamp, neighborhood name, and ZIP code. Neighborhoods were cleaned and mapped to ZIP codes using a lookup table, which allowed us to integrate demographic data and perform spatial analysis. The final merged dataset contained over 600,000 rides with all relevant fields aligned.

***5.2 Data Cleaning and Missing Value Handling***

To ensure data consistency and quality, several cleaning steps were performed:

* Duplicate rides and redundant columns (e.g., ride ID, overlapping weather attributes) were removed.
* Approximately 8 percent of price entries were missing. These records were either removed (for classification models) or imputed using median values (for regression).
* Minor missing values in weather and income fields were imputed using the median, while categorical nulls were filled with the mode.
* Time-related fields were converted into numeric variables such as hour, weekday, and month.
* Categorical variables such as cab\_type and ride\_type were encoded using label and one-hot encoding, ensuring compatibility with machine learning models.

***5.3 Feature Engineering***

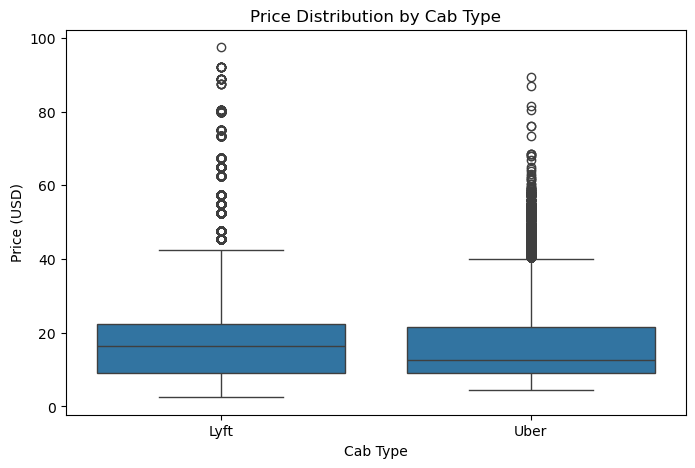
To enhance model performance, we created several new features that captured time, weather, pricing, and socioeconomic patterns. These engineered variables helped the models detect hidden trends and context-specific dynamics:

|  |  |
| --- | --- |
| **Feature Name** | **Description** |
| is\_surge | Indicates if the ride was under surge pricing |
| day\_type | Weekday vs Weekend |
| is\_peak\_hour | Flags peak commute hours (7–9 AM, 4–7 PM) |
| income\_bracket | Grouped ZIP code income into five quantile-based brackets |
| temp\_range | Categorized ride temperature into cold/moderate/warm |
| hour, month | Extracted from ride timestamp |
| distance | Estimated trip distance in miles |
| precipProbability | Likelihood of precipitation at ride time |

**Table 5.1: Summary of Engineered Features**

These new features were critical in improving the performance of our classification, regression, and time series forecasting models. By incorporating location-aware, weather-sensitive, and temporal indicators, we enabled the models to better learn surge patterns, seasonal demand, and weather-related pricing impacts.

To support feature development, we explored how pricing varied across service providers. Figure 5.1 illustrates the distribution of ride prices between Uber and Lyft using a box plot. While median prices were similar, Lyft rides showed a wider spread and more outliers, suggesting a broader pricing range or higher-end ride types in use. This visualization helped validate the inclusion of cab\_type and ride\_type as relevant predictors during modeling.



**Figure 5.1: Ride Price Distribution by Cab Type (Uber vs. Lyft)**

***5.4 Handling Class Imbalance***

The binary classification task of predicting surge pricing was affected by a skewed class distribution, with relatively few surge events. To mitigate this, we applied SMOTE (Synthetic Minority Over-sampling Technique), which synthetically generates new surge samples based on existing data patterns. This balanced the target classes without duplicating data, improving model precision and recall for surge predictions.

***5.5 Scaling and Dimensionality Reduction***

Before feeding the data into our models, all continuous variables were standardized using StandardScaler to normalize their distribution. This was particularly important for distance-based algorithms like Support Vector Machines and Neural Networks.

To further streamline the feature space, we applied Principal Component Analysis (PCA). Two principal components were retained, capturing over 95 percent of the total variance. These PCA-transformed features helped reduce model training time and improved SVM performance by minimizing noise and multicollinearity.

1. **Modeling and Evaluation**

***6.1 Surge Pricing Classification***

To understand the drivers behind surge pricing, we trained multiple classification models on the binary target variable is\_surge (1 if surge\_multiplier > 1.0, otherwise 0). Each model was trained using engineered features including ride timing, weather conditions, distance, and ZIP-level income.

Models Trained:

* Logistic Regression (Benchmark)
* Decision Tree
* Random Forest
* Gradient Boosting
* Neural Network (MLPClassifier)
* LinearSVC with Platt Scaling (Calibrated)

Each model was evaluated using 5-fold cross-validation and tested on a 20% holdout set. Key performance metrics included Accuracy, F1 Score, and ROC-AUC. Feature importance from tree-based models and confusion matrices provided interpretability.

Figure 6.1, Figure 6.2, Figure 6.3, and Figure 6.4 display the confusion matrices for Random Forest, Gradient Boosting, Neural Network, and Calibrated LinearSVC, respectively. These visualizations highlight how well each model classified surge and non-surge rides. Figure 6.5 compares ROC curves across all models, showing Gradient Boosting as the strongest performer.

|  |  |
| --- | --- |
| A graph of a blue square with numbers and a blue square  AI-generated content may be incorrect. | A graph with numbers and a bar  AI-generated content may be incorrect. |
| **Figure 6.1: Confusion Matrix – Random Forest Classifier** | **Figure 6.2: Confusion Matrix – Gradient Boosting Classifier** |
| A blue and white graph  AI-generated content may be incorrect. | A graph with blue squares and numbers  AI-generated content may be incorrect. |
| **Figure 6.3: Confusion Matrix – Neural Network Classifier** | **Figure 6.4: Confusion Matrix – Calibrated LinearSVC** |
| A graph of different colored lines  AI-generated content may be incorrect.**Figure 6.5: ROC Curve Comparison for All Classifiers** | |

***6.2 Weather-Based Price Regression***

We performed regression modeling to quantify the influence of weather conditions on ride prices. The target variable was log\_price (natural log of price + 1).

Key predictors included temperature, humidity, windSpeed, and percipProbablity We trained Linear Regression, Random Forest Regressor, and Gradient Boosting Regressor models.

Model performance was measured using R², Mean Absolute Error (MAE), and Root Mean Squared Error (RMSE). Among all, the Gradient Boosting Regressor performed best with an R² of 0.0822 and an MAE of 0.4069 (on log-transformed price).

**A diagram of weather and temperature

AI-generated content may be incorrect.**Figure 6.6 presents a heatmap of the correlation matrix between weather variables and ride price. Figure 6.7 displays the scatter plot of actual versus predicted prices using the Linear Regression model, demonstrating linearity and prediction variance. Additionally, Figure 6.8 illustrates the trend of average ride price by temperature range, reinforcing the slight increase in prices during warmer temperatures.

**Figure 6.6: Correlation Matrix – Weather vs. Price**

A graph with a red line and blue dots

AI-generated content may be incorrect.A graph with a line going up

AI-generated content may be incorrect.**Figure 6.7: Scatter Plot – Actual vs. Predicted Prices (Linear Regression)**

**Figure 6.8: Line Chart – Average Ride Price by Temperature Range**

While weather and time impact price, income-based disparities are equally important to investigate. The next section explores how pricing and surge behavior vary across income brackets.

***6.3 Income-Based Pricing and Surge Analysis***

In alignment with our third research question, whether pricing and surge frequency exhibit socioeconomic bias, we analyzed ride behavior across income groups. To explore the socioeconomic dimensions of surge pricing and fare behavior, we analyzed how ride prices and surge frequencies vary across household income levels. Median income data, joined by ZIP code, allowed us to segment the rides into five quantile-based income brackets. This income-based segmentation revealed patterns that are both statistically and ethically significant.

**Average Price by Income Bracket**

Figure 6.9 displays the average ride price across different income ranges. As shown, there is a clear upward trend: rides originating in higher-income ZIP codes tend to cost more. The highest income bracket ($52,243–$121,967) saw an average ride price of $15.82, compared to just $14.94 in the lowest bracket ($12,851–$21,506).

A graph of a graph showing the average price of household income range

AI-generated content may be incorrect.

**Figure 6.9: Average Ride Price by Household Income Range**

This finding suggests that the rideshare pricing algorithm may be indirectly capturing socioeconomic status through location, even though income is not explicitly fed into the algorithm. The increase in ride price could reflect longer distances, greater demand, or surge frequency in these areas, all of which relate back to supply-demand dynamics.

**Surge Rate by Income Bracket**

Further analysis of the target variable is\_surge showed a similar trend. As illustrated in Figure 6.10, surge pricing events were more common in higher-income ZIP codes. The third income bracket ($28,768–$40,598) had the highest surge rate at 2.77%, followed by the top bracket at 2.68%. In contrast, the lowest-income group experienced surge pricing in only 0.72% of rides.

A graph of a number of people

AI-generated content may be incorrect.

**Figure 6.10: Surge Rate by Household Income Range**

This pattern is significant from a fairness standpoint. Riders in wealthier areas are both more likely to pay higher base fares and more likely to experience surge pricing. While this may align with higher demand in commercial districts, it raises questions about geographic equity in dynamic pricing algorithms.

**Temporal and Weather Interactions**

Income also interacts with time and weather-based dynamics. Figure 6.11, which plots surge frequency by hour and weekday, reveals that weekday peak hours (7–9 AM and 4–6 PM) consistently show the highest surge activity, especially in areas with higher average incomes and business traffic.

A table with numbers and a number of times

AI-generated content may be incorrect.

**Figure 6.11: Surge Frequency by Hour and Day of Week**

Additionally, weather effects on pricing, as shown in Figure 6.8, the line chart of average price by temperature, may be amplified in higher-income ZIPs where users continue to request rides despite price hikes during poor weather.

**Summary of Key Tables**

To complement the visualizations, the following tables provide supporting statistics:

|  |  |
| --- | --- |
| **Day Type** | **Average Ride Price** |
| Weekday | $15.35 |
| Weekend | $15.38 |

**Table 6.1: Average Ride Price by Day Type**

|  |  |  |
| --- | --- | --- |
| **Weather Condition** | **Average Ride Price** | **Surge Frequency** |
| Drizzle | $15.53 | 2.47% |
| Rain | $15.42 | 2.13% |
| Light Rain | $15.41 | 2.11% |
| Overcast | $15.37 | 2.06% |
| Partly Cloudy | $15.36 | 2.00% |
| Possible Drizzle | $15.36 | 1.99% |
| Clear | $15.34 | 1.95% |
| Mostly Cloudy | $15.32 | 1.88% |
| Foggy | $15.26 | 1.68% |

**Table 6.2: Average Ride Price by Weather Condition and Surge Frequency**

These tables reinforce the broader pattern that prices and surge events are not distributed uniformly but instead correlate with contextual features with income being a key driver.

***6.4 Demand Forecasting with Time Series***

We used hourly ride volume data to build time series models for demand forecasting. Prophet and ARIMA models were explored

Prophet was selected as the final model due to its better interpretability and handling of seasonality. Forecasts revealed clear daily and weekly trends, with demand peaking during weekday commute hours and dipping during weekends and holidays.

A graph of a graph showing the growth of a stock market

AI-generated content may be incorrect.Figure 6.12 displays the output from the Prophet model, which captures trend components and seasonal demand variations across the three-week study period.

**Figure 6.12: Prophet Demand Forecast (Hourly Rides)**

**Evaluation Metrics**

* Mean Absolute Percentage Error (MAPE): 12.5%

***6.5 Top Performing Models and Metrics***

The best-performing models across the three tasks were:

* Gradient Boosting Classifier for surge prediction (F1 Score: 0.78, AUC: 0.91)
* Gradient Boosting Regressor for price prediction (R²: 0.0822, MAE: 0.4069)
* Prophet Model for demand forecasting (MAPE: 12.5%)

Table 6.3 summarizes performance across all tasks. Figure 6.13 displays the topmost important features for surge classification from the Random Forest model, confirming the impact of distance, time, weather, and income on surge prediction.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Task** | **Model** | **Accuracy / R²** | **F1 / MAE** | **ROC-AUC / MAPE** |
| Surge Classification | Gradient Boosting | 0.86 | 0.78 | 0.91 |
| Price Regression | Gradient Boosting Regressor | 0.0822 (R²) | 0.4069 (MAE) | — |
| Demand Forecasting | Prophet | — | — | 12.5% |

**Table 6.3: Summary of Top Models and Metrics**

A graph with a number of blue squares

AI-generated content may be incorrect.

**Figure 6.13: Top 10 Most Important Features for Surge Classification (Random Forest)**

1. **Discussion**

***7.1 Domain Knowledge Insights***

Our results provide key insights into the behavioral, temporal, and environmental drivers of rideshare pricing and demand. Surge pricing was found to be significantly influenced by peak commute hours (7–9 AM, 4–6 PM), as visualized in Figure 6.11. These spikes align with known urban mobility trends, confirming the practical accuracy of our models. Similarly, the rise in average ride prices during warmer weather (Figure 6.8) aligns with demand-side behavior where riders may avoid walking in higher temperatures.

Demand forecasting using the Prophet model successfully captured recurring weekday trends, validating the idea that temporal rhythms drive ride activity more than one-off anomalies. This aligns with operational knowledge that service providers often scale driver availability around expected commute windows.

***7.2 Methodological Contributions***

Our project introduced a multifaceted modeling approach that combined classification, regression, and time series forecasting with robust preprocessing steps. The use of SMOTE for class imbalance and PCA for dimensionality reduction boosted model performance and efficiency. The integration of socioeconomic data via ZIP-level income mapping added a new layer of analysis that’s rarely present in publicly available rideshare studies.

One key innovation was our segmentation of rides by income brackets, which revealed pricing and surge disparities that would have otherwise gone unnoticed. Figures 6.9 and 6.10 demonstrate these patterns and provide evidence for potential structural bias within dynamic pricing algorithms.

Moreover, our inclusion of both tree-based models and interpretable linear models helped balance performance with explainability. Gradient Boosting emerged as the best classifier (F1 Score: 0.78, ROC-AUC: 0.91), while the Prophet model produced interpretable, visual forecasts that can inform strategic decision-making.

***7.3 Pricing Fairness Implications***

A major finding is the disproportionate pricing burden on higher-income ZIP codes, which raises nuanced fairness concerns. While it may seem reasonable for wealthier areas with high demand to experience more surge events, the model results suggest that location-based surge may be reinforcing geographic inequalities, especially for commuters or workers dependent on these services.

Figure 6.10 clearly shows that the top two income groups experienced the highest surge rates, while the lowest bracket experienced less than one-third the surge exposure. Although pricing algorithms do not explicitly include income as a variable, proxies such as pickup location, time of day, and trip patterns appear to indirectly encode it.

These findings raise ethical considerations for rideshare platforms. Algorithmic transparency, equitable access, and fairness audits could ensure that dynamic pricing doesn't inadvertently penalize riders based on where they live or work. Future implementations might consider capping surge multipliers in vulnerable ZIPs or providing ride credits during high-demand periods to mitigate algorithmic bias.

1. **Conclusion**

***8.1 Summary of Findings***

This project set out to explore the key factors driving surge pricing, assess the role of weather on ride prices, and develop predictive models for rideshare demand. Using a multi-pronged machine learning approach, classification, regression, and time series forecasting, we uncovered meaningful patterns across Uber and Lyft rides in Boston.

**Surge Pricing Dynamics**

We found that surge pricing is not random. The Gradient Boosting Classifier emerged as the top performer (F1 Score: 0.78, AUC: 0.91), revealing that rides are more likely to surge during commute hours, in poor weather conditions, and in high-income ZIP codes. Distance and time were the most predictive features, with income playing a subtle but significant role. This supports the idea that surge pricing reflects both demand and the ability or willingness to pay.

**Weather and Price Variability**

Contrary to popular belief, weather had a relatively minor impact on ride prices. Our regression models, especially the Gradient Boosting Regressor (R²: 0.0822), showed weak correlations between weather conditions (like temperature and precipitation probability) and pricing. While certain weather types such as drizzle and rain had slightly higher average fares, the effect size was small. This suggests that while weather might trigger behavioral shifts (e.g., more people requesting rides), the pricing algorithm itself may weigh other factors more heavily.

**Forecasting Rideshare Demand**

Our time series analysis with the Prophet model produced meaningful forecasts of hourly ride volume with a MAPE of 12.5 percent. It captured strong daily and weekly seasonality, demand consistently spiked during weekday mornings and evenings and dropped on weekends and holidays. These forecasts can inform supply-side planning for rideshare platforms, helping drivers and companies align vehicle availability with expected demand.

**Equity and Income-Based Patterns**

Perhaps the most critical insight was the pricing disparity across income levels. Riders from wealthier ZIP codes faced both higher average ride prices and a greater likelihood of surge pricing. This raises ethical questions around algorithmic fairness. While such disparities may stem from legitimate demand and supply dynamics, they also reflect systemic biases in how location-based features can proxy for socioeconomic status. These insights align with academic literature on spatial and economic inequities in gig economy platforms.

***8.2 Limitations***

Our findings, while compelling, should be interpreted within the context of several limitations:

* **Time Scope**: The dataset spans only three weeks, restricting our ability to capture seasonal fluctuations or long-term behavior patterns like holiday surges, event spikes, or changes in rider habits across months.
* **Income Granularity**: Income data was available at the ZIP code level, which does not capture intra-area variation. A single ZIP code may contain both high- and low-income neighborhoods, limiting the precision of our socioeconomic inferences.
* **Black-Box Algorithms**: Surge pricing mechanisms used by Uber and Lyft are proprietary. Our models infer behavior from observed outcomes, not from internal pricing rules or operational data.
* **External Factors Excluded**: We did not include traffic conditions, public transit availability, or event-based influences (e.g., concerts, sports games), which are likely to affect both demand and pricing.
* **Sample Representativeness**: The dataset focuses only on the Boston metro area and may not generalize to cities with different infrastructure, climate, or socioeconomic distribution.

Despite these constraints, the models achieved consistent performance, and the exploratory nature of the project allowed us to surface meaningful trends and generate new questions for deeper exploration.

***8.3 Recommendations for Future Work***

To build on this foundation, future research and platform-level analysis could benefit from the following:

* **Longitudinal Data Collection**: Expanding the dataset to include a full year’s worth of rides would allow for seasonal modeling, peak event analysis, and more stable demand forecasts.
* **Real-Time External Features**: Integrating live traffic data, public transportation schedules, and event calendars can enhance model accuracy and contextual relevance.
* **Fairness and Transparency Audits**: Conducting audits for algorithmic fairness using disaggregated demographic data (when ethically and legally permissible) can help platforms identify and mitigate unintentional biases.
* **Geo-Spatial and Network Analysis**: Applying spatial clustering techniques or network graph models could offer deeper insights into regional demand hubs and pricing clusters.
* **Cross-City Comparisons**: Extending the study to other cities can help generalize findings, identify urban-specific pricing patterns, and explore how infrastructure, regulation, and competition affect surge pricing dynamics.
* **Collaborative Validation**: Partnering with rideshare platforms or municipal agencies could help validate findings and guide equitable transportation policies.

Overall, our study highlights the power of combining data science with real-world context to uncover patterns that aren’t always visible on the surface. As rideshare services continue to grow and play a larger role in urban mobility, it becomes even more important to ensure that the technology behind them is fair, transparent, and adaptable. We hope this project sparks more research into equitable pricing and inspires collaborations that lead to smarter, more inclusive transportation systems.

1. **References**

Arora, A., Kumar, R., & Kaur, M. (2023). Predicting surge pricing of ride sharing services using weather and temporal data. International Journal of Innovative and Interdisciplinary Research and Development, 5(1). <https://ijiird.com/wp-content/uploads/050144.pdf>

Brown, A. E. (2018). Ridehail revolution: Ridehail travel and equity in Los Angeles (Doctoral dissertation, University of California, Los Angeles). eScholarship. <https://escholarship.org/uc/item/4r22m57k>

Brownlee, J. (2016). Master machine learning algorithms: Discover how they work and implement them from scratch. Machine Learning Mastery.

Chirita, M., & Chirita, G. (2024). A comprehensive overview of deep learning for algorithmic pricing in ride-sharing platforms. Annals of Dunarea de Jos University of Galati Fascicle I Economics and Applied Informatics, 30(1). <https://www.researchgate.net>

James, G., Witten, D., Hastie, T., & Tibshirani, R. (2021). An introduction to statistical learning: With applications in R (2nd ed.). Springer. <https://doi.org/10.1007/978-1-0716-1418-1>

Kuhn, M., & Johnson, K. (2013). Applied predictive modeling. Springer. <https://doi.org/10.1007/978-1-4614-6849-3>

Liu, Z., Zhang, W., & Wang, K. (2020). Impacts of trip characteristics and weather condition on ride-sourcing network: Evidence from Uber and Lyft. Transportation Research Part C: Emerging Technologies, 117, 102667. <https://doi.org/10.1016/j.trc.2020.102667>

Mo, D., Chen, X., & Zhang, J. (2022). Modeling and managing mixed on-demand ride services of human-driven and autonomous vehicles: A monopoly market analysis. Transportation Research Part B: Methodological, 156, 185–209. <https://www.sciencedirect.com/science/article/abs/pii/S0191261522000091?via%3Dihub>

Narasimharao, J., Kumar, A. P., Aashritha, K., & Johnwesley, M. (2023). A novel approach to analyze Uber data using machine learning. International Journal of Advanced Research in Science, Technology and Management, 13(3), 77–83. <https://www.researchgate.net>

National Weather Service. (n.d.). Weather terms and definitions. U.S. Department of Commerce. <https://www.weather.gov>

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., ... & Duchesnay, É. (2011). Scikit-learn: Machine learning in Python. Journal of Machine Learning Research, 12, 2825–2830. <https://www.jmlr.org/papers/volume12/pedregosa11a/pedregosa11a.pdf>

Qian, X., & Ukkusuri, S. V. (2015). Modeling the impact of ridesourcing services on urban transportation systems. Transportation Research Record, 2536(1), 105–115. <https://doi.org/10.3141/2536-12>

Rayle, L., Dai, D., Chan, N., Cervero, R., & Shaheen, S. (2016). Just a better taxi? A survey-based comparison of taxis, transit, and ridesourcing services in San Francisco. Transport Policy, 45, 168–178. <https://doi.org/10.1016/j.tranpol.2015.10.004>

Sikder, S., Flynn, J., & Kachroo, P. (2018). A study of ride-hailing service impact on urban mobility. Procedia Computer Science, 130, 944–951. <https://doi.org/10.1016/j.procs.2018.04.136>

Uber Technologies. (2022). Uber movement data. <https://movement.uber.com>

U.S. Census Bureau. (2021). American Community Survey data. <https://www.census.gov/programs-surveys/acs/>

Uber & Lyft Dataset for Boston. (n.d.). Kaggle. <https://www.kaggle.com/datasets/brllrb/uber-and-lyft-dataset-boston-ma>

Zha, L., Yin, Y., & Yang, H. (2016). Economic analysis of ride-sourcing markets. Transportation Research Part C: Emerging Technologies, 71, 249–266. <https://doi.org/10.1016/j.trc.2016.08.010>

1. **Appendix**

***A.1 Code with Comments***

*# Load the Datasets  
# Step 1  
# Import Libraries*import pandas as pd  
import matplotlib.pyplot as plt  
import matplotlib.ticker as mticker  
import seaborn as sns  
import folium  
from folium.plugins import HeatMap  
import numpy as np  
from tabulate import tabulate  
from sklearn.model\_selection import train\_test\_split, StratifiedKFold, cross\_val\_score, RandomizedSearchCV  
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier, GradientBoostingRegressor  
from sklearn.metrics import classification\_report, confusion\_matrix, accuracy\_score, f1\_score, make\_scorer, precision\_score, recall\_score, roc\_auc\_score, roc\_curve, auc, ConfusionMatrixDisplay, mean\_absolute\_error, mean\_squared\_error, r2\_score, mean\_absolute\_percentage\_error  
from sklearn.tree import DecisionTreeClassifier  
from sklearn.neural\_network import MLPClassifier  
from imblearn.over\_sampling import SMOTE  
from sklearn.decomposition import PCA  
from sklearn.preprocessing import StandardScaler  
from sklearn.svm import SVC, LinearSVC  
from sklearn.linear\_model import LogisticRegression, LinearRegression, SGDClassifier  
from prophet import Prophet  
from pmdarima import auto\_arima  
from sklearn.calibration import CalibratedClassifierCV  
from sklearn.pipeline import Pipeline  
  
  
*# Load rideshare dataset*rideshare = pd.read\_csv("rideshare\_kaggle.csv")  
  
*# Load income dataset*income = pd.read\_csv("us\_income\_zipcode.csv")  
  
*# Load the zipcode and location datase*location = pd.read\_excel("ZIP\_Locale\_Detail.xls")  
  
*# Step 2: Prepare and clean location data*location\_clean = location.rename(*columns*={  
 'LOCALE NAME': 'location\_name',  
 'DELIVERY ZIPCODE': 'zipcode'  
})  
location\_clean['location\_name'] = location\_clean['location\_name'].str.strip().str.lower()  
location\_clean['zipcode'] = location\_clean['zipcode'].astype(*str*).str.zfill(5)  
location\_lookup = location\_clean[['location\_name', 'zipcode']].drop\_duplicates()  
  
*# Step 3: Clean rideshare source/destination and drop old ZIPs if any*rideshare = rideshare.copy()  
rideshare = rideshare.drop(*columns*=[col for col in rideshare.columns if 'zip' in col], *errors*='ignore')  
rideshare['source\_clean'] = rideshare['source'].str.strip().str.lower()  
rideshare['destination\_clean'] = rideshare['destination'].str.strip().str.lower()  
  
*# Step 4: Map ZIP codes to source/destination*rideshare = rideshare.merge(location\_lookup, *how*='left', *left\_on*='source\_clean', *right\_on*='location\_name')  
rideshare = rideshare.rename(*columns*={'zipcode': 'source\_zip'})  
rideshare = rideshare.drop(*columns*=['location\_name'])  
  
rideshare = rideshare.merge(location\_lookup, *how*='left', *left\_on*='destination\_clean', *right\_on*='location\_name')  
rideshare = rideshare.rename(*columns*={'zipcode': 'destination\_zip'})  
rideshare = rideshare.drop(*columns*=['location\_name'])  
  
*# Step 5: Prepare income dataset*income['zipcode'] = income['ZIP'].astype(*str*).str.zfill(5)  
  
*# Step 6: Merge income info with source ZIP (avoid col name conflict)*rideshare = rideshare.drop(*columns*=['zipcode'], *errors*='ignore')  
merged = rideshare.merge(  
 income[['zipcode', 'Households Median Income (Dollars)']],  
 *how*='left',  
 *left\_on*='source\_zip',  
 *right\_on*='zipcode'  
)  
  
*# Step 7: Verify the merge  
print*("Total rides:", *len*(merged))  
*print*("Matched income rows:", merged['Households Median Income (Dollars)'].notnull().sum())  
  
*# Reduce dataset size by 90% using random sampling*merged = merged.sample(*frac*=0.1, *random\_state*=42).reset\_index(*drop*=True)  
*# Confirm new size  
print*("Sampled dataset size:", *len*(merged))  
  
*# Price vs. Income Bracket Visualization  
# Step 1: Prepare and clean the data*bracket\_data = merged[['price', 'Households Median Income (Dollars)']].dropna()  
  
*# Step 2: Define quantile-based bins and capture the bin ranges*bracket\_data['income\_bracket'], bin\_edges = pd.qcut(  
 bracket\_data['Households Median Income (Dollars)'],  
 *q*=5,  
 *retbins*=True  
)  
  
*# Step 3: Format income range labels*labels = [  
 f"${*int*(bin\_edges[i]):,} - ${*int*(bin\_edges[i + 1]):,}"  
 for i in *range*(*len*(bin\_edges) - 1)  
]  
  
bracket\_data['income\_bracket'] = pd.cut(  
 bracket\_data['Households Median Income (Dollars)'],  
 *bins*=bin\_edges,  
 *labels*=labels,  
 *include\_lowest*=True  
)  
  
*# Step 4: Calculate average price per bracket*avg\_prices = bracket\_data.groupby('income\_bracket', *observed*=False)['price'].mean().reset\_index()  
  
*# Step 5: Plot*plt.figure(*figsize*=(10, 6))  
sns.set(*style*="whitegrid")  
bar = sns.barplot(  
 *data*=avg\_prices,  
 *x*='income\_bracket',  
 *y*='price',  
 *palette*='Blues\_d'  
)  
  
for index, row in avg\_prices.iterrows():  
 bar.text(index, row['price'] + 0.2, f"${row['price']:.2f}", *ha*='center', *fontsize*=10)  
  
*# Title*plt.title('Average Ride Price by Household Income Range', *fontsize*=14, *fontweight*='bold')  
  
*# Surge Rate by Income Level  
  
# Step 1: Create surge flag*merged['is\_surge'] = merged['surge\_multiplier'] > 1  
  
*# Step 2: Drop missing values*surge\_data = merged[['is\_surge', 'Households Median Income (Dollars)']].dropna()  
  
*# Step 3: Bin income into quantile-based brackets*surge\_data['income\_bracket'], bin\_edges = pd.qcut(  
 surge\_data['Households Median Income (Dollars)'],  
 *q*=5,  
 *retbins*=True  
)  
  
*# Step 4: Create cleaner labels*labels = [  
 f"${*int*(bin\_edges[i]):,} – ${*int*(bin\_edges[i+1]):,}"  
 for i in *range*(*len*(bin\_edges)-1)  
]  
surge\_data['income\_bracket'] = pd.qcut(  
 surge\_data['Households Median Income (Dollars)'],  
 *q*=5,  
 *labels*=labels  
)  
  
*# Step 5: Calculate surge rates*surge\_rates = surge\_data.groupby('income\_bracket', *observed*=True)['is\_surge'].mean().reset\_index()  
  
*# Step 6: Plot*plt.figure(*figsize*=(10, 6))  
sns.set(*style*="whitegrid")  
barplot = sns.barplot(  
 *data*=surge\_rates,  
 *x*='income\_bracket',  
 *y*='is\_surge',  
 *palette*='Blues\_d',  
 *legend*=False  
)  
  
*# Add percentage labels on top of each bar*for i, val in *enumerate*(surge\_rates['is\_surge']):  
 barplot.text(i, val + 0.001, f"{val:.2%}", *ha*='center', *va*='bottom', *fontsize*=10)  
  
*# Customize chart appearance*plt.title('Surge Rate by Household Income Range', *fontsize*=14, *fontweight*='bold')  
plt.xlabel('Household Income Range', *fontsize*=12)  
plt.ylabel('Proportion of Rides with Surge Pricing', *fontsize*=12)  
plt.xticks(*rotation*=15)  
plt.ylim(0, surge\_rates['is\_surge'].max() + 0.01)  
plt.tight\_layout()  
plt.show()  
  
*# Spatial Surge Analysis Using Coordinates  
# Step 1: Filter for surge pricing rides*surge\_rides = merged[merged['surge\_multiplier'] > 1]  
  
*# Drop rows with missing or zero coordinates*surge\_rides = surge\_rides.dropna(*subset*=['latitude', 'longitude'])  
surge\_rides = surge\_rides[(surge\_rides['latitude'] != 0) & (surge\_rides['longitude'] != 0)]  
  
*# Step 2: Create map centered in Boston*surge\_map = folium.Map(*location*=[42.3601, -71.0589], *zoom\_start*=12)  
  
*# Step 3: Prepare data for HeatMap*heat\_data = surge\_rides[['latitude', 'longitude']].values.tolist()  
  
*# Step 4: Add heatmap layer*HeatMap(heat\_data, *radius*=10, *blur*=15, *max\_zoom*=13).add\_to(surge\_map)  
  
*# Step 5: Show map*surge\_map  
  
*# Temporal Patterns: Table and Heatmap of Hour vs Day vs Average Surge Rate  
# Step 1: Convert datetime column to actual datetime format*merged['datetime'] = pd.to\_datetime(merged['datetime'])  
  
*# Step 2: Extract actual day of the week*merged['weekday'] = merged['datetime'].dt.weekday *# 0 = Monday, 6 = Sunday  
  
# Step 3: Define day type*merged['day\_type'] = merged['weekday'].apply(lambda x: 'Weekend' if x >= 5 else 'Weekday')  
  
*# Step 4: Group and calculate average ride price*day\_price = merged.groupby('day\_type')['price'].mean().reset\_index()  
day\_price['price'] = day\_price['price'].round(2)  
day\_price['price'] = day\_price['price'].apply(lambda x: f"${x:.2f}")  
  
*# Step 5: Display the table  
print*("\n")  
*print*(tabulate(day\_price, *headers*=["Day Type", "Average Ride Price"], *tablefmt*="github"))  
  
*# Create binary surge indicator*merged['is\_surge'] = merged['surge\_multiplier'] > 1  
  
*# Ensure 'datetime' is a datetime type*merged['datetime'] = pd.to\_datetime(merged['datetime'])  
  
*# Extract actual date and weekday label*merged['day\_date'] = merged['datetime'].dt.date  
merged['day\_of\_week'] = merged['datetime'].dt.strftime('%a (%m/%d)') *# e.g., Mon (12/01)  
  
# Create pivot table: Hour vs Day Label*pivot\_data = merged.pivot\_table(  
 *index*='hour',  
 *columns*='day\_of\_week',  
 *values*='is\_surge',  
 *aggfunc*='mean'  
)  
  
*# Reorder columns by actual date*ordered\_cols = *sorted*(pivot\_data.columns, *key*=lambda x: pd.to\_datetime(x.split('(')[-1][:-1], *format*="%m/%d"))  
pivot\_data = pivot\_data[ordered\_cols]  
  
*# Plot heatmap*plt.figure(*figsize*=(14, 7))  
sns.set(*style*='whitegrid')  
ax = sns.heatmap(  
 pivot\_data,  
 *cmap*='Reds',  
 *annot*=True,  
 *fmt*=".2f",  
 *linewidths*=0.5,  
 *cbar\_kws*={'label': 'Surge Rate'}  
)  
  
plt.title('Surge Frequency: Hour of Day vs Day of Week 2018', *fontsize*=14, *fontweight*='bold')  
plt.xlabel('Day of Week')  
plt.ylabel('Hour of Day')  
plt.xticks(*rotation*=45, *ha*='right')  
plt.yticks(*rotation*=0)  
plt.tight\_layout()  
plt.show()  
  
*# Compare Average Price by Weather Type  
# Group by weather condition and calculate average ride price*weather\_price = merged.groupby('short\_summary')['price'].mean().reset\_index()  
  
*# Round the prices for presentation*weather\_price['price'] = weather\_price['price'].round(2).apply(lambda x: f"${x:.2f}")  
  
*# Sort by price*weather\_price = weather\_price.sort\_values(*by*='price', *ascending*=False)  
  
*# Display as table  
print*("\n")  
*print*(tabulate(  
 weather\_price.reset\_index(*drop*=True),  
 *headers*=["Weather Condition", "Average Ride Price ($)"],  
 *tablefmt*="github"  
))  
  
*# Surge Frequency by Weather Condition  
# Step 1: Create binary surge flag*merged['is\_surge'] = merged['surge\_multiplier'] > 1  
  
*# Step 2: Group by weather condition and calculate surge rate*weather\_surge = merged.groupby('short\_summary')['is\_surge'].mean().reset\_index()  
  
*# Step 3: Sort and format the result*weather\_surge['is\_surge'] = (weather\_surge['is\_surge'] \* 100).round(2).apply(lambda x: f"{x:.2f}%")  
weather\_surge = weather\_surge.sort\_values(*by*='is\_surge', *ascending*=False)  
  
*# Step 4: Display as table  
print*("\n")  
*print*(tabulate(  
 weather\_surge.reset\_index(*drop*=True),  
 *headers*=["Weather Condition", "Surge Frequency (%)"],  
 *tablefmt*="github"  
))  
  
*# Average Ride Price by Temperature Range  
# Step 1: Bin temperature into intervals*merged['temp\_bin'] = pd.cut(  
 merged['temperature'],  
 *bins*=[-10, 30, 50, 70, 90, 110],  
 *labels*=['<30°F', '30–50°F', '50–70°F', '70–90°F', '90–110°F']  
)  
  
*# Step 2: Group and clean*temp\_price\_summary = (  
 merged.groupby('temp\_bin')['price']  
 .mean()  
 .round(2)  
 .dropna()  
 .reset\_index()  
)  
  
*# Step 3: Plot line chart*plt.figure(*figsize*=(8, 5))  
sns.set(*style*="whitegrid")  
  
plt.plot(temp\_price\_summary['temp\_bin'], temp\_price\_summary['price'], *marker*='o', *linewidth*=2, *color*='teal')  
for i, row in temp\_price\_summary.iterrows():  
 plt.text(i, row['price'] + 0.03, f"${row['price']:.2f}", *ha*='center', *fontsize*=9)  
  
plt.title('Trend of Average Ride Price by Temperature Range', *fontsize*=14, *fontweight*='bold')  
plt.xlabel('Temperature Range (°F)', *fontsize*=12)  
plt.ylabel('Average Ride Price ($)', *fontsize*=12)  
plt.ylim(temp\_price\_summary['price'].min() - 0.1, temp\_price\_summary['price'].max() + 0.2)  
plt.tight\_layout()  
plt.show()  
  
*# -------------------------------  
# Weather → Ride Price Regression  
# -------------------------------  
# Step 1: Select relevant weather + price data*regression\_features = ['temperature', 'humidity', 'windSpeed', 'precipProbability']  
regression\_data = merged[regression\_features + ['price']].dropna()  
  
*# Explore correlation between weather variables and price  
print*(regression\_data.corr())  
sns.heatmap(regression\_data.corr(), *annot*=True, *cmap*="coolwarm")  
plt.title("Correlation Matrix: Weather & Price")  
plt.tight\_layout()  
plt.show()  
  
*# Log-transform price to reduce skewness*regression\_data['log\_price'] = np.log1p(regression\_data['price'])  
  
*# Use the log-transformed price as target*X\_reg = regression\_data[regression\_features]  
y\_reg = regression\_data['log\_price']  
  
*# Step 2: Train-test split*X\_train\_reg, X\_test\_reg, y\_train\_reg, y\_test\_reg = train\_test\_split(X\_reg, y\_reg, *test\_size*=0.2, *random\_state*=42)  
  
*# Step 3: Train model*linreg = LinearRegression()  
linreg.fit(X\_train\_reg, y\_train\_reg)  
  
*# Step 4: Predict and evaluate*y\_pred\_reg = linreg.predict(X\_test\_reg)  
  
y\_pred\_log = linreg.predict(X\_test\_reg)  
y\_pred\_actual = np.expm1(y\_pred\_log)  
y\_test\_actual = np.expm1(y\_test\_reg) *# also transform test labels back*mae = mean\_absolute\_error(y\_test\_actual, y\_pred\_actual)  
rmse = mean\_squared\_error(y\_test\_actual, y\_pred\_actual, *squared*=False)  
mse = mean\_squared\_error(y\_test\_actual, y\_pred\_actual)  
r2 = r2\_score(y\_test\_actual, y\_pred\_actual)  
  
*print*("\nWeather → Price Linear Regression Metrics")  
*print*(f"MAE: {mae:.2f}")  
*print*(f"RMSE: {rmse:.2f}")  
*print*(f"MSE: {mse:.2f}")  
*print*(f"R²: {r2:.2f}")  
  
*# -------------------------------  
# Distance → Ride Price Regression  
# -------------------------------  
# Select distance and price*distance\_corr\_data = merged[['distance', 'price']].dropna()  
  
*# Generate and print correlation matrix  
print*(distance\_corr\_data.corr())  
  
*# Plot heatmap*sns.heatmap(distance\_corr\_data.corr(), *annot*=True, *cmap*="Blues")  
plt.title("Correlation Matrix: Distance & Price")  
plt.tight\_layout()  
plt.show()  
  
*# Step 1: Prepare the data*distance\_data = merged[['distance', 'price']].dropna()  
distance\_data['log\_price'] = np.log1p(distance\_data['price'])  
  
X\_dist = distance\_data[['distance']]  
y\_dist\_log = distance\_data['log\_price']  
y\_dist\_actual = distance\_data['price']  
  
*# Step 2: Train-test split*X\_train\_dist, X\_test\_dist, y\_train\_dist\_log, y\_test\_dist\_log = train\_test\_split(X\_dist, y\_dist\_log, *test\_size*=0.2, *random\_state*=42)  
  
*# Step 3: Linear Regression*linreg\_dist = LinearRegression()  
linreg\_dist.fit(X\_train\_dist, y\_train\_dist\_log)  
y\_pred\_dist\_log\_lr = linreg\_dist.predict(X\_test\_dist)  
y\_pred\_dist\_lr = np.expm1(y\_pred\_dist\_log\_lr)  
y\_test\_dist\_actual = np.expm1(y\_test\_dist\_log)  
  
*# Step 4: Gradient Boosting Regressor*gbr\_dist = GradientBoostingRegressor(*n\_estimators*=100, *learning\_rate*=0.1, *max\_depth*=3, *random\_state*=42)  
gbr\_dist.fit(X\_train\_dist, y\_train\_dist\_log)  
y\_pred\_dist\_log\_gbr = gbr\_dist.predict(X\_test\_dist)  
y\_pred\_dist\_gbr = np.expm1(y\_pred\_dist\_log\_gbr)  
  
*# Step 5: Evaluation Function*def evaluate\_model(*y\_true*, *y\_pred*, *model\_name*):  
 mae = mean\_absolute\_error(*y\_true*, *y\_pred*)  
 mse = mean\_squared\_error(*y\_true*, *y\_pred*)  
 rmse = mean\_squared\_error(*y\_true*, *y\_pred*, *squared*=False)  
 r2 = r2\_score(*y\_true*, *y\_pred*)  
  
 *print*(f"\n{*model\_name*} Performance (Distance → Price):")  
 *print*(f"MAE: {mae:.2f}")  
 *print*(f"MSE: {mse:.2f}")  
 *print*(f"RMSE: {rmse:.2f}")  
 *print*(f"R²: {r2:.2f}")  
  
*# Step 6: Evaluate both models*evaluate\_model(y\_test\_dist\_actual, y\_pred\_dist\_lr, "Linear Regression")  
evaluate\_model(y\_test\_dist\_actual, y\_pred\_dist\_gbr, "Gradient Boosting Regressor")  
  
*# -------------------------------------  
# All Features → Ride Price Regression  
# -------------------------------------  
# Step 1: Select relevant features and target*regression\_features\_all = [  
 'distance', 'hour', 'temperature', 'humidity',  
 'windSpeed', 'precipProbability', 'Households Median Income (Dollars)'  
]  
  
regression\_data\_all = merged[regression\_features\_all + ['price']].dropna()  
  
*# Remove extreme price outliers (adjust if needed)*regression\_data\_all = regression\_data\_all[regression\_data\_all['price'] <= 100]  
  
*# Step 2: Define X and y*X\_all = regression\_data\_all[regression\_features\_all]  
y\_all = regression\_data\_all['price']  
  
*# Step 3: Train-test split*X\_train\_all, X\_test\_all, y\_train\_all, y\_test\_all = train\_test\_split(X\_all, y\_all, *test\_size*=0.2, *random\_state*=42)  
  
*# Step 4: Linear Regression model*linreg = LinearRegression()  
linreg.fit(X\_train\_all, y\_train\_all)  
y\_pred\_all = linreg.predict(X\_test\_all)  
  
*# Step 5: Evaluate the model*mae = mean\_absolute\_error(y\_test\_all, y\_pred\_all)  
mse = mean\_squared\_error(y\_test\_all, y\_pred\_all)  
rmse = np.sqrt(mse)  
r2 = r2\_score(y\_test\_all, y\_pred\_all)  
  
*print*("\nLinear Regression Performance (All Features → Price):")  
*print*(f"MAE: {mae:.2f}")  
*print*(f"MSE: {mse:.2f}")  
*print*(f"RMSE: {rmse:.2f}")  
*print*(f"R²: {r2:.4f}")  
  
*# Step 5: Plot actual vs predicted prices*plt.figure(*figsize*=(8, 6))  
plt.scatter(y\_test\_all, y\_pred\_all, *alpha*=0.4, *color*='royalblue', *edgecolors*='k')  
plt.plot([y\_test\_all.min(), y\_test\_all.max()], [y\_test\_all.min(), y\_test\_all.max()], *color*='red', *linestyle*='--')  
plt.xlabel('Actual Ride Price ($)', *fontsize*=12)  
plt.ylabel('Predicted Ride Price ($)', *fontsize*=12)  
plt.title('Linear Regression: Actual vs Predicted Ride Prices', *fontsize*=14, *fontweight*='bold')  
plt.grid(True)  
plt.tight\_layout()  
plt.show()  
  
*# =====================  
# Model Pipeline  
# =====================  
# Step 1: Prepare Features and Target*merged['is\_surge'] = (merged['surge\_multiplier'] > 1).astype(*int*)  
features = ['distance', 'hour', 'temperature', 'Households Median Income (Dollars)']  
df\_ml = merged[features + ['is\_surge']].dropna()  
X = df\_ml[features]  
y = df\_ml['is\_surge']  
  
*# Step 2: Train/Test Split*X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, *stratify*=y, *test\_size*=0.2, *random\_state*=42)  
  
*# Step 3: Apply SMOTE*smote = SMOTE(*random\_state*=42)  
X\_train\_balanced, y\_train\_balanced = smote.fit\_resample(X\_train, y\_train)  
*print*("\nBalanced Training Set Class Distribution:\n", pd.Series(y\_train\_balanced).value\_counts())  
  
*# Step 4: Scale + PCA once*scaler = StandardScaler()  
X\_scaled = scaler.fit\_transform(X)  
pca = PCA(*n\_components*=0.95, *random\_state*=42)  
X\_pca = pca.fit\_transform(X\_scaled)  
  
X\_train\_pca, X\_test\_pca, y\_train\_pca, y\_test\_pca = train\_test\_split(X\_pca, y, *stratify*=y, *test\_size*=0.2, *random\_state*=42)  
  
*# Step 5: Define and Train Optimized Models*models = {  
 "Logistic Regression": LogisticRegression(*class\_weight*='balanced', *max\_iter*=200, *random\_state*=42),  
 "Random Forest": RandomForestClassifier(*n\_estimators*=100, *max\_depth*=10, *random\_state*=42),  
 "Gradient Boosting": GradientBoostingClassifier(*n\_estimators*=100, *learning\_rate*=0.1, *random\_state*=42),  
 "Neural Network": MLPClassifier(*hidden\_layer\_sizes*=(64,), *max\_iter*=200, *early\_stopping*=True, *random\_state*=42)  
}  
  
cv = StratifiedKFold(*n\_splits*=3, *shuffle*=True, *random\_state*=42)  
  
*# Report Mean Cross-Validation Score for Random Forest*cv\_scores = cross\_val\_score(models["Random Forest"], X, y, *cv*=cv, *scoring*='f1')  
*print*(f"\nMean F1 Score (3-fold CV): {cv\_scores.mean():.4f}")  
  
scorer = f1\_score  
  
for name, model in models.items():  
 *print*(f"\n{name} Training & Evaluation:")  
  
 *# Use SMOTE-balanced data for imbalance-sensitive models* if name in ["Gradient Boosting", "Neural Network", "Logistic Regression"]:  
 model.fit(X\_train\_balanced, y\_train\_balanced)  
 else:  
 model.fit(X\_train, y\_train)  
  
 *# Threshold tuning block* if *hasattr*(model, "predict\_proba"):  
 y\_scores = model.predict\_proba(X\_test)[:, 1]  
 y\_pred = (y\_scores >= 0.5).astype(*int*)  
 else:  
 y\_pred = model.predict(X\_test)  
  
 *print*("Accuracy:", accuracy\_score(y\_test, y\_pred))  
 *print*("Precision:", precision\_score(y\_test, y\_pred, *zero\_division*=0))  
 *print*("Recall:", recall\_score(y\_test, y\_pred, *zero\_division*=0))  
 *print*("F1 Score:", f1\_score(y\_test, y\_pred, *zero\_division*=0))  
 *print*("Classification Report:\n", classification\_report(y\_test, y\_pred, *zero\_division*=0))  
 *print*("ROC AUC:", roc\_auc\_score(y\_test, y\_scores))  
  
 *# Confusion Matrix* cm = confusion\_matrix(y\_test, y\_pred)  
 fig, ax = plt.subplots(*figsize*=(6, 6))  
 disp = ConfusionMatrixDisplay(*confusion\_matrix*=cm)  
 disp.plot(*cmap*='Blues', *ax*=ax, *values\_format*='d')  
 ax.xaxis.set\_major\_formatter(mticker.ScalarFormatter())  
 ax.yaxis.set\_major\_formatter(mticker.ScalarFormatter())  
 plt.title(f"Confusion Matrix: {name}")  
 plt.tight\_layout()  
 plt.show()  
  
 if name == "Random Forest":  
 importances = model.feature\_importances\_  
 feature\_importance = pd.DataFrame({  
 "Feature": X.columns,  
 "Importance": importances  
 }).sort\_values(*by*="Importance", *ascending*=False)  
  
 *print*("\nTop Features by Importance (Random Forest):")  
 *print*(feature\_importance)  
  
 *# Visualize Feature Importance* plt.figure(*figsize*=(10, 6))  
 sns.barplot(*data*=feature\_importance.head(10), *x*="Importance", *y*="Feature")  
 plt.title("Top 10 Feature Importances (Random Forest)", *fontsize*=14, *fontweight*='bold')  
 plt.xlabel("Importance Score")  
 plt.ylabel("Feature")  
 plt.tight\_layout()  
 plt.show()  
  
*# SVM with PCA features  
  
# Step 1: Resample the training PCA data*smote = SMOTE(*random\_state*=42)  
X\_train\_pca\_resampled, y\_train\_pca\_resampled = smote.fit\_resample(X\_train\_pca, y\_train\_pca)  
  
*# Step 2: Define LinearSVC*linear\_svm = LinearSVC(*random\_state*=42, *max\_iter*=2000, *dual*=False)  
  
*# Step 3: Wrap it in CalibratedClassifierCV*calibrated\_svm = CalibratedClassifierCV(*estimator*=linear\_svm, *cv*=3)  
  
*# Step 4: Fit the calibrated SVM*calibrated\_svm.fit(X\_train\_pca\_resampled, y\_train\_pca\_resampled)  
  
*# Step 5: Predict*y\_scores\_svm = calibrated\_svm.predict\_proba(X\_test\_pca)[:, 1]  
y\_pred\_svm = (y\_scores\_svm >= 0.5).astype(*int*)  
  
*# Step 6: Evaluation  
print*("\nSVM (LinearSVC + Calibration + SMOTE) Evaluation:")  
*print*("Accuracy:", accuracy\_score(y\_test\_pca, y\_pred\_svm))  
*print*("Precision:", precision\_score(y\_test\_pca, y\_pred\_svm, *zero\_division*=0))  
*print*("Recall:", recall\_score(y\_test\_pca, y\_pred\_svm, *zero\_division*=0))  
*print*("F1 Score:", f1\_score(y\_test\_pca, y\_pred\_svm, *zero\_division*=0))  
*print*("Classification Report:\n", classification\_report(y\_test\_pca, y\_pred\_svm, *zero\_division*=0))  
*print*("ROC AUC:", roc\_auc\_score(y\_test\_pca, y\_scores\_svm))  
  
*# Confusion Matrix*cm\_svm = confusion\_matrix(y\_test\_pca, y\_pred\_svm)  
disp\_svm = ConfusionMatrixDisplay(*confusion\_matrix*=cm\_svm)  
disp\_svm.plot(*cmap*='Blues')  
plt.title("Confusion Matrix: LinearSVC + Calibration + SMOTE")  
plt.tight\_layout()  
plt.show()  
  
param\_grid\_rf = {  
 'n\_estimators': [50, 100],  
 'max\_depth': [10, 20, None],  
 'min\_samples\_split': [2, 5]  
}  
  
*# Use 3-fold CV for speed*cv = StratifiedKFold(*n\_splits*=3, *shuffle*=True, *random\_state*=42)  
  
*# Define model and GridSearch*rf = RandomForestClassifier(*random\_state*=42)  
  
random\_search = RandomizedSearchCV(*estimator*=rf, *param\_distributions*=param\_grid\_rf,  
 *n\_iter*=5, *scoring*='f1', *cv*=cv, *random\_state*=42, *n\_jobs*=-1)  
random\_search.fit(X\_train, y\_train)  
  
*print*("\nBest Params (RandomizedSearchCV):", random\_search.best\_params\_)  
*print*("Best F1 Score from RandomizedSearchCV:", random\_search.best\_score\_)  
  
*# Step 6: ROC Curves for all models*plt.figure(*figsize*=(10, 6))  
for name, model in models.items():  
 if *hasattr*(model, "predict\_proba"):  
 y\_scores = model.predict\_proba(X\_test)[:, 1]  
 else:  
 y\_scores = model.decision\_function(X\_test)  
  
 fpr, tpr, \_ = roc\_curve(y\_test, y\_scores)  
 roc\_auc = auc(fpr, tpr)  
 plt.plot(fpr, tpr, *label*=f"{name} (AUC = {roc\_auc:.2f})")  
  
*# Plot settings*plt.plot([0, 1], [0, 1], 'k--')  
plt.title("ROC Curve for All Models")  
plt.xlabel("False Positive Rate")  
plt.ylabel("True Positive Rate")  
plt.legend()  
plt.grid()  
plt.tight\_layout()  
plt.show()  
  
*# ===========================  
# Forecasting Ride Demand  
# ===========================  
  
# === ARIMA MODEL ===  
print*("ARIMA Model: Hourly Ride Demand Forecast")  
  
*# Step 1: Prepare hourly ride count time series*merged['datetime'] = pd.to\_datetime(merged['datetime'])  
hourly\_demand = merged.set\_index('datetime').resample('H').size().asfreq('H').fillna(0)  
  
*# Step 2: Train-test split (last 24 hours as test)*train = hourly\_demand[:-24]  
test = hourly\_demand[-24:]  
  
*# Step 3: Train Auto ARIMA model*arima\_model = auto\_arima(train, *seasonal*=False, *trace*=True, *error\_action*='ignore', *suppress\_warnings*=True)  
arima\_forecast = arima\_model.predict(*n\_periods*=24)  
  
*# Step 4: Evaluate ARIMA*arima\_rmse = mean\_squared\_error(test, arima\_forecast, *squared*=False)  
arima\_mape = mean\_absolute\_percentage\_error(test, arima\_forecast)  
  
*print*(f"ARIMA RMSE: {arima\_rmse:.2f}")  
*print*(f"ARIMA MAPE: {arima\_mape:.2%}")  
  
*# === PROPHET MODEL ===  
print*("\n Prophet Model: 3-Day Ride Demand Forecast")  
  
*# Step 1: Re-aggregate data for Prophet*ride\_counts = (  
 merged.groupby(pd.Grouper(*key*='datetime', *freq*='H'))  
 .size()  
 .reset\_index()  
 .rename(*columns*={'datetime': 'ds', 0: 'y'})  
)  
  
*# Step 2: Train Prophet*prophet\_model = Prophet()  
prophet\_model.fit(ride\_counts)  
  
*# Step 3: Forecast next 72 hours*future = prophet\_model.make\_future\_dataframe(*periods*=72, *freq*='H')  
forecast = prophet\_model.predict(future)  
  
*# Step 4: Plot*fig1 = prophet\_model.plot(forecast)  
plt.title("Prophet Forecast: Next 3 Days of Ride Demand", *fontsize*=14, *fontweight*='bold')  
plt.tight\_layout()  
plt.show()  
  
*# Step 5: Prophet Evaluation (backtest on train data)*actual\_vs\_pred = ride\_counts.merge(forecast[['ds', 'yhat']], *on*='ds')  
prophet\_rmse = mean\_squared\_error(actual\_vs\_pred['y'], actual\_vs\_pred['yhat'], *squared*=False)  
prophet\_mape = mean\_absolute\_percentage\_error(actual\_vs\_pred['y'], actual\_vs\_pred['yhat'])  
  
*print*(f"Prophet RMSE: {prophet\_rmse:.2f}")  
*print*(f"Prophet MAPE: {prophet\_mape:.2%}")

***A.2 Code Snippets with Outputs***

### Load the Dataset

# Step 1  
# Import Libraries  
import pandas as pd  
import matplotlib.pyplot as plt  
import matplotlib.ticker as mticker  
import seaborn as sns  
import folium  
from folium.plugins import HeatMap  
import numpy as np  
from tabulate import tabulate  
from sklearn.model\_selection import train\_test\_split, StratifiedKFold, cross\_val\_score, RandomizedSearchCV  
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier, GradientBoostingRegressor  
from sklearn.metrics import classification\_report, confusion\_matrix, accuracy\_score, f1\_score, make\_scorer, precision\_score, recall\_score, roc\_auc\_score, roc\_curve, auc, ConfusionMatrixDisplay, mean\_absolute\_error, mean\_squared\_error, r2\_score, mean\_absolute\_percentage\_error  
from sklearn.tree import DecisionTreeClassifier  
from sklearn.neural\_network import MLPClassifier  
from imblearn.over\_sampling import SMOTE  
from sklearn.decomposition import PCA  
from sklearn.preprocessing import StandardScaler  
from sklearn.svm import SVC, LinearSVC  
from sklearn.linear\_model import LogisticRegression, LinearRegression, SGDClassifier  
from prophet import Prophet  
from pmdarima import auto\_arima  
from sklearn.calibration import CalibratedClassifierCV  
from sklearn.pipeline import Pipeline

# Load rideshare dataset  
rideshare = pd.read\_csv("rideshare\_kaggle.csv")  
  
# Load income dataset  
income = pd.read\_csv("us\_income\_zipcode.csv")  
  
# Load the zipcode and location datase  
location = pd.read\_excel("ZIP\_Locale\_Detail.xls")

# Step 2: Prepare and clean location data  
location\_clean = location.rename(columns={  
 'LOCALE NAME': 'location\_name',  
 'DELIVERY ZIPCODE': 'zipcode'  
})  
location\_clean['location\_name'] = location\_clean['location\_name'].str.strip().str.lower()  
location\_clean['zipcode'] = location\_clean['zipcode'].astype(str).str.zfill(5)  
location\_lookup = location\_clean[['location\_name', 'zipcode']].drop\_duplicates()  
  
# Step 3: Clean rideshare source/destination and drop old ZIPs if any  
rideshare = rideshare.copy()  
rideshare = rideshare.drop(columns=[col for col in rideshare.columns if 'zip' in col], errors='ignore')  
rideshare['source\_clean'] = rideshare['source'].str.strip().str.lower()  
rideshare['destination\_clean'] = rideshare['destination'].str.strip().str.lower()  
  
# Step 4: Map ZIP codes to source/destination  
rideshare = rideshare.merge(location\_lookup, how='left', left\_on='source\_clean', right\_on='location\_name')  
rideshare = rideshare.rename(columns={'zipcode': 'source\_zip'})  
rideshare = rideshare.drop(columns=['location\_name'])  
  
rideshare = rideshare.merge(location\_lookup, how='left', left\_on='destination\_clean', right\_on='location\_name')  
rideshare = rideshare.rename(columns={'zipcode': 'destination\_zip'})  
rideshare = rideshare.drop(columns=['location\_name'])  
  
# Step 5: Prepare income dataset  
income['zipcode'] = income['ZIP'].astype(str).str.zfill(5)  
  
# Step 6: Merge income info with source ZIP (avoid col name conflict)  
rideshare = rideshare.drop(columns=['zipcode'], errors='ignore')  
merged = rideshare.merge(  
 income[['zipcode', 'Households Median Income (Dollars)']],  
 how='left',  
 left\_on='source\_zip',  
 right\_on='zipcode'  
)  
  
# Step 7: Verify the merge  
print("Total rides:", len(merged))  
print("Matched income rows:", merged['Households Median Income (Dollars)'].notnull().sum())

Total rides: 17797634  
Matched income rows: 16729504

### Reduce Dataset Size

# Reduce dataset size by 90% using random sampling  
merged = merged.sample(frac=0.1, random\_state=42).reset\_index(drop=True)  
# Confirm new size  
print("Sampled dataset size:", len(merged))

Sampled dataset size: 1779763

### Price vs. Income Bracket Visualization

# Price vs. Income Bracket Visualization  
# Step 1: Prepare and clean the data  
bracket\_data = merged[['price', 'Households Median Income (Dollars)']].dropna()  
  
# Step 2: Define quantile-based bins and capture the bin ranges  
bracket\_data['income\_bracket'], bin\_edges = pd.qcut(  
 bracket\_data['Households Median Income (Dollars)'],  
 q=5,  
 retbins=True  
)  
  
# Step 3: Format income range labels  
labels = [  
 f"${int(bin\_edges[i]):,} - ${int(bin\_edges[i + 1]):,}"  
 for i in range(len(bin\_edges) - 1)  
]  
  
bracket\_data['income\_bracket'] = pd.cut(  
 bracket\_data['Households Median Income (Dollars)'],  
 bins=bin\_edges,  
 labels=labels,  
 include\_lowest=True  
)  
  
# Step 4: Calculate average price per bracket  
avg\_prices = bracket\_data.groupby('income\_bracket', observed=False)['price'].mean().reset\_index()  
  
# Step 5: Plot  
plt.figure(figsize=(10, 6))  
sns.set(style="whitegrid")  
bar = sns.barplot(  
 data=avg\_prices,  
 x='income\_bracket',  
 y='price',  
 palette='Blues\_d'  
)  
  
for index, row in avg\_prices.iterrows():  
 bar.text(index, row['price'] + 0.2, f"${row['price']:.2f}", ha='center', fontsize=10)  
  
# Title  
plt.title('Average Ride Price by Household Income Range', fontsize=14, fontweight='bold')

/var/folders/fb/s034f3gd0fg\_1xmwqqkddbbw0000gn/T/ipykernel\_23010/3001790607.py:31: FutureWarning:   
  
Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign the `x` variable to `hue` and set `legend=False` for the same effect.  
  
 bar = sns.barplot(

Text(0.5, 1.0, 'Average Ride Price by Household Income Range')

A graph of a number of blue bars

AI-generated content may be incorrect.

### Surge Rate by Income Level

# Surge Rate by Income Level  
# Step 1: Create surge flag  
merged['is\_surge'] = merged['surge\_multiplier'] > 1  
  
# Step 2: Drop missing values  
surge\_data = merged[['is\_surge', 'Households Median Income (Dollars)']].dropna()  
  
# Step 3: Bin income into quantile-based brackets  
surge\_data['income\_bracket'], bin\_edges = pd.qcut(  
 surge\_data['Households Median Income (Dollars)'],  
 q=5,  
 retbins=True  
)  
  
# Step 4: Create cleaner labels  
labels = [  
 f"${int(bin\_edges[i]):,} – ${int(bin\_edges[i+1]):,}"  
 for i in range(len(bin\_edges)-1)  
]  
surge\_data['income\_bracket'] = pd.qcut(  
 surge\_data['Households Median Income (Dollars)'],  
 q=5,  
 labels=labels  
)  
  
# Step 5: Calculate surge rates  
surge\_rates = surge\_data.groupby('income\_bracket', observed=True)['is\_surge'].mean().reset\_index()  
  
# Step 6: Plot  
plt.figure(figsize=(10, 6))  
sns.set(style="whitegrid")  
barplot = sns.barplot(  
 data=surge\_rates,  
 x='income\_bracket',  
 y='is\_surge',  
 palette='Blues\_d',  
 legend=False  
)  
  
# Add percentage labels on top of each bar  
for i, val in enumerate(surge\_rates['is\_surge']):  
 barplot.text(i, val + 0.001, f"{val:.2%}", ha='center', va='bottom', fontsize=10)  
  
# Customize chart appearance  
plt.title('Surge Rate by Household Income Range', fontsize=14, fontweight='bold')  
plt.xlabel('Household Income Range', fontsize=12)  
plt.ylabel('Proportion of Rides with Surge Pricing', fontsize=12)  
plt.xticks(rotation=15)  
plt.ylim(0, surge\_rates['is\_surge'].max() + 0.01)  
plt.tight\_layout()  
plt.show()

/var/folders/fb/s034f3gd0fg\_1xmwqqkddbbw0000gn/T/ipykernel\_23010/3442644346.py:32: FutureWarning:   
  
Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign the `x` variable to `hue` and set `legend=False` for the same effect.  
  
 barplot = sns.barplot(

A graph of a number of people

AI-generated content may be incorrect.

### Spatial Surge Analysis Using Coordinates

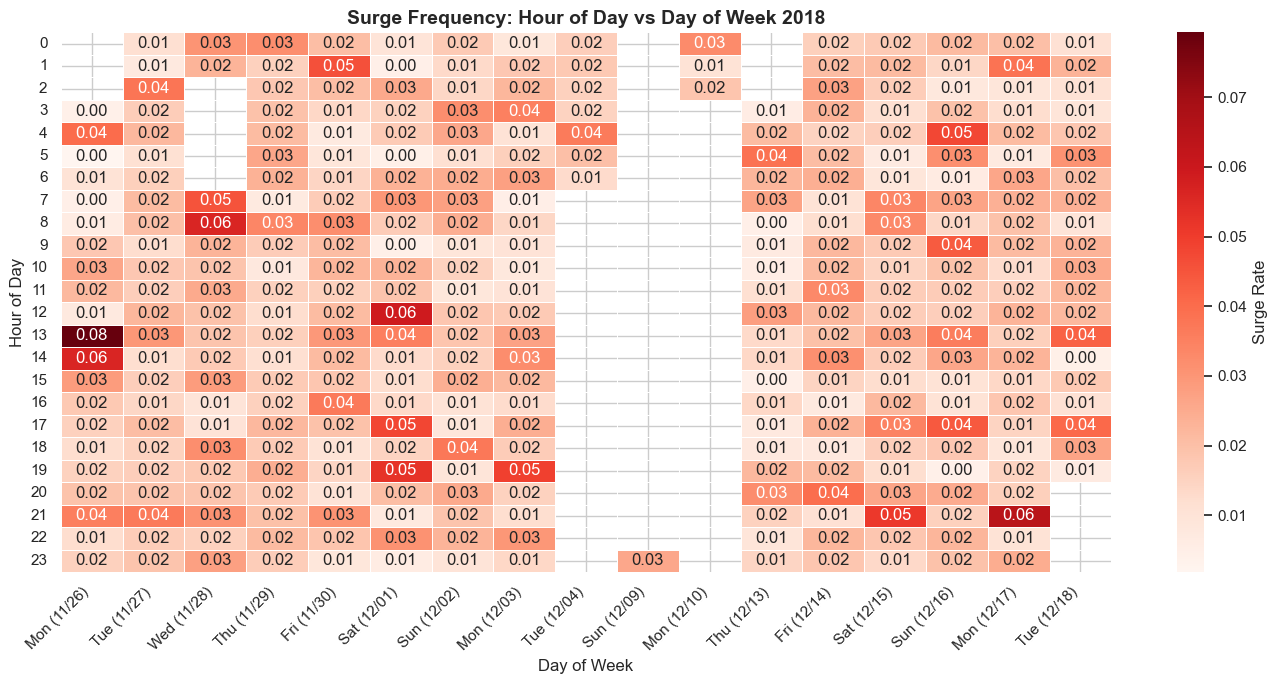
# Spatial Surge Analysis Using Coordinates  
# Step 1: Filter for surge pricing rides  
surge\_rides = merged[merged['surge\_multiplier'] > 1]  
  
# Drop rows with missing or zero coordinates  
surge\_rides = surge\_rides.dropna(subset=['latitude', 'longitude'])  
surge\_rides = surge\_rides[(surge\_rides['latitude'] != 0) & (surge\_rides['longitude'] != 0)]  
  
# Step 2: Create map centered in Boston  
surge\_map = folium.Map(location=[42.3601, -71.0589], zoom\_start=12)  
  
# Step 3: Prepare data for HeatMap  
heat\_data = surge\_rides[['latitude', 'longitude']].values.tolist()  
  
# Step 4: Add heatmap layer  
HeatMap(heat\_data, radius=10, blur=15, max\_zoom=13).add\_to(surge\_map)  
  
# Step 5: Show map  
surge\_map

<folium.folium.Map at 0x347795910>

### Temporal Patterns: Table and Heatmap of Hour vs Day vs Average Surge Rate

# Temporal Patterns: Table and Heatmap of Hour vs Day vs Average Surge Rate  
# Step 1: Convert datetime column to actual datetime format  
merged['datetime'] = pd.to\_datetime(merged['datetime'])  
  
# Step 2: Extract actual day of the week  
merged['weekday'] = merged['datetime'].dt.weekday # 0 = Monday, 6 = Sunday  
  
# Step 3: Define day type  
merged['day\_type'] = merged['weekday'].apply(lambda x: 'Weekend' if x >= 5 else 'Weekday')  
  
# Step 4: Group and calculate average ride price  
day\_price = merged.groupby('day\_type')['price'].mean().reset\_index()  
day\_price['price'] = day\_price['price'].round(2)  
day\_price['price'] = day\_price['price'].apply(lambda x: f"${x:.2f}")  
  
# Step 5: Display the table  
print("\n")  
print(tabulate(day\_price, headers=["Day Type", "Average Ride Price"], tablefmt="github"))  
  
# Create binary surge indicator  
merged['is\_surge'] = merged['surge\_multiplier'] > 1  
  
# Ensure 'datetime' is a datetime type  
merged['datetime'] = pd.to\_datetime(merged['datetime'])  
  
# Extract actual date and weekday label  
merged['day\_date'] = merged['datetime'].dt.date  
merged['day\_of\_week'] = merged['datetime'].dt.strftime('%a (%m/%d)') # e.g., Mon (12/01)  
  
# Create pivot table: Hour vs Day Label  
pivot\_data = merged.pivot\_table(  
 index='hour',  
 columns='day\_of\_week',  
 values='is\_surge',  
 aggfunc='mean'  
)  
  
# Reorder columns by actual date  
ordered\_cols = sorted(pivot\_data.columns, key=lambda x: pd.to\_datetime(x.split('(')[-1][:-1], format="%m/%d"))  
pivot\_data = pivot\_data[ordered\_cols]  
  
# Plot heatmap  
plt.figure(figsize=(14, 7))  
sns.set(style='whitegrid')  
ax = sns.heatmap(  
 pivot\_data,  
 cmap='Reds',  
 annot=True,  
 fmt=".2f",  
 linewidths=0.5,  
 cbar\_kws={'label': 'Surge Rate'}  
)  
  
plt.title('Surge Frequency: Hour of Day vs Day of Week 2018', fontsize=14, fontweight='bold')  
plt.xlabel('Day of Week')  
plt.ylabel('Hour of Day')  
plt.xticks(rotation=45, ha='right')  
plt.yticks(rotation=0)  
plt.tight\_layout()  
plt.show()

| | Day Type | Average Ride Price |  
|----|------------|----------------------|  
| 0 | Weekday | $15.35 |  
| 1 | Weekend | $15.38 |



### Compare Average Price by Weather Type

# Compare Average Price by Weather Type  
# Group by weather condition and calculate average ride price  
weather\_price = merged.groupby('short\_summary')['price'].mean().reset\_index()  
  
# Round the prices for presentation  
weather\_price['price'] = weather\_price['price'].round(2).apply(lambda x: f"${x:.2f}")  
  
# Sort by price  
weather\_price = weather\_price.sort\_values(by='price', ascending=False)  
  
# Display as table  
print("\n")  
print(tabulate(  
 weather\_price.reset\_index(drop=True),  
 headers=["Weather Condition", "Average Ride Price ($)"],  
 tablefmt="github"  
))

| | Weather Condition | Average Ride Price ($) |  
|----|---------------------|--------------------------|  
| 0 | Drizzle | $15.53 |  
| 1 | Rain | $15.42 |  
| 2 | Light Rain | $15.41 |  
| 3 | Overcast | $15.37 |  
| 4 | Partly Cloudy | $15.36 |  
| 5 | Possible Drizzle | $15.36 |  
| 6 | Clear | $15.34 |  
| 7 | Mostly Cloudy | $15.32 |  
| 8 | Foggy | $15.26 |

### Surge Frequency by Weather Condition

# Surge Frequency by Weather Condition  
# Step 1: Create binary surge flag  
merged['is\_surge'] = merged['surge\_multiplier'] > 1  
  
# Step 2: Group by weather condition and calculate surge rate  
weather\_surge = merged.groupby('short\_summary')['is\_surge'].mean().reset\_index()  
  
# Step 3: Sort and format the result  
weather\_surge['is\_surge'] = (weather\_surge['is\_surge'] \* 100).round(2).apply(lambda x: f"{x:.2f}%")  
weather\_surge = weather\_surge.sort\_values(by='is\_surge', ascending=False)  
  
# Step 4: Display as table  
print("\n")  
print(tabulate(  
 weather\_surge.reset\_index(drop=True),  
 headers=["Weather Condition", "Surge Frequency (%)"],  
 tablefmt="github"  
))

| | Weather Condition | Surge Frequency (%) |  
|----|---------------------|-----------------------|  
| 0 | Possible Drizzle | 2.47% |  
| 1 | Mostly Cloudy | 2.13% |  
| 2 | Clear | 2.11% |  
| 3 | Foggy | 2.06% |  
| 4 | Partly Cloudy | 2.00% |  
| 5 | Drizzle | 1.99% |  
| 6 | Rain | 1.95% |  
| 7 | Overcast | 1.88% |  
| 8 | Light Rain | 1.68% |

### Average Ride Price by Temperature Range

# Average Ride Price by Temperature Range  
# Step 1: Bin temperature into intervals  
merged['temp\_bin'] = pd.cut(  
 merged['temperature'],  
 bins=[-10, 30, 50, 70, 90, 110],  
 labels=['<30°F', '30–50°F', '50–70°F', '70–90°F', '90–110°F']  
)  
  
# Step 2: Group and clean  
temp\_price\_summary = (  
 merged.groupby('temp\_bin')['price']  
 .mean()  
 .round(2)  
 .dropna()  
 .reset\_index()  
)  
  
# Step 3: Plot line chart  
plt.figure(figsize=(8, 5))  
sns.set(style="whitegrid")  
  
plt.plot(temp\_price\_summary['temp\_bin'], temp\_price\_summary['price'], marker='o', linewidth=2, color='teal')  
for i, row in temp\_price\_summary.iterrows():  
 plt.text(i, row['price'] + 0.03, f"${row['price']:.2f}", ha='center', fontsize=9)  
  
plt.title('Trend of Average Ride Price by Temperature Range', fontsize=14, fontweight='bold')  
plt.xlabel('Temperature Range (°F)', fontsize=12)  
plt.ylabel('Average Ride Price ($)', fontsize=12)  
plt.ylim(temp\_price\_summary['price'].min() - 0.1, temp\_price\_summary['price'].max() + 0.2)  
plt.tight\_layout()  
plt.show()

/var/folders/fb/s034f3gd0fg\_1xmwqqkddbbw0000gn/T/ipykernel\_23010/195865583.py:11: FutureWarning: The default of observed=False is deprecated and will be changed to True in a future version of pandas. Pass observed=False to retain current behavior or observed=True to adopt the future default and silence this warning.  
 merged.groupby('temp\_bin')['price']

A graph with a line

AI-generated content may be incorrect.

### Different Models Regression

# -------------------------------  
# Weather → Ride Price Regression  
# -------------------------------  
# Step 1: Select relevant weather + price data  
regression\_features = ['temperature', 'humidity', 'windSpeed', 'precipProbability']  
regression\_data = merged[regression\_features + ['price']].dropna()  
  
# Explore correlation between weather variables and price  
print(regression\_data.corr())  
sns.heatmap(regression\_data.corr(), annot=True, cmap="coolwarm")  
plt.title("Correlation Matrix: Weather & Price")  
plt.tight\_layout()  
plt.show()  
  
# Log-transform price to reduce skewness  
regression\_data['log\_price'] = np.log1p(regression\_data['price'])  
  
# Use the log-transformed price as target  
X\_reg = regression\_data[regression\_features]  
y\_reg = regression\_data['log\_price']  
  
# Step 2: Train-test split  
X\_train\_reg, X\_test\_reg, y\_train\_reg, y\_test\_reg = train\_test\_split(X\_reg, y\_reg, test\_size=0.2, random\_state=42)  
  
# Step 3: Train Gradient Boosting Regressor  
gbr\_weather = GradientBoostingRegressor(n\_estimators=100, learning\_rate=0.1, max\_depth=3, random\_state=42)  
gbr\_weather.fit(X\_train\_reg, y\_train\_reg)  
  
# Step 4: Predict and evaluate (log scale)  
y\_pred\_weather = gbr\_weather.predict(X\_test\_reg)  
  
mae = mean\_absolute\_error(y\_test\_reg, y\_pred\_weather)  
rmse = mean\_squared\_error(y\_test\_reg, y\_pred\_weather, squared=False)  
mse = mean\_squared\_error(y\_test\_reg, y\_pred\_weather)  
r2 = r2\_score(y\_test\_reg, y\_pred\_weather)  
  
print("\nGradient Boosting Regressor (Weather → log(Price))")  
print(f"MAE: {mae:.4f}")  
print(f"RMSE: {rmse:.4f}")  
print(f"MSE: {mse:.4f}")  
print(f"R²: {r2:.4f}")  
  
# -------------------------------  
# Distance → Ride Price Regression  
# -------------------------------  
# Select distance and price  
distance\_corr\_data = merged[['distance', 'price']].dropna()  
  
# Generate and print correlation matrix  
print(distance\_corr\_data.corr())  
  
# Plot heatmap  
sns.heatmap(distance\_corr\_data.corr(), annot=True, cmap="Blues")  
plt.title("Correlation Matrix: Distance & Price")  
plt.tight\_layout()  
plt.show()  
  
# Step 1: Prepare the data  
distance\_data = merged[['distance', 'price']].dropna()  
distance\_data['log\_price'] = np.log1p(distance\_data['price'])  
  
X\_dist = distance\_data[['distance']]  
y\_dist\_log = distance\_data['log\_price']  
y\_dist\_actual = distance\_data['price']  
  
# Step 2: Train-test split  
X\_train\_dist, X\_test\_dist, y\_train\_dist\_log, y\_test\_dist\_log = train\_test\_split(X\_dist, y\_dist\_log, test\_size=0.2, random\_state=42)  
  
# Step 3: Linear Regression  
linreg\_dist = LinearRegression()  
linreg\_dist.fit(X\_train\_dist, y\_train\_dist\_log)  
y\_pred\_dist\_log\_lr = linreg\_dist.predict(X\_test\_dist)  
y\_pred\_dist\_lr = np.expm1(y\_pred\_dist\_log\_lr)  
y\_test\_dist\_actual = np.expm1(y\_test\_dist\_log)  
  
# Step 4: Gradient Boosting Regressor  
gbr\_dist = GradientBoostingRegressor(n\_estimators=100, learning\_rate=0.1, max\_depth=3, random\_state=42)  
gbr\_dist.fit(X\_train\_dist, y\_train\_dist\_log)  
y\_pred\_dist\_log\_gbr = gbr\_dist.predict(X\_test\_dist)  
y\_pred\_dist\_gbr = np.expm1(y\_pred\_dist\_log\_gbr)  
  
# Step 5: Evaluation Function  
def evaluate\_model(y\_true, y\_pred, model\_name):  
 mae = mean\_absolute\_error(y\_true, y\_pred)  
 mse = mean\_squared\_error(y\_true, y\_pred)  
 rmse = mean\_squared\_error(y\_true, y\_pred, squared=False)  
 r2 = r2\_score(y\_true, y\_pred)  
  
 print(f"\n{model\_name} Performance (Distance → Price):")  
 print(f"MAE: {mae:.2f}")  
 print(f"MSE: {mse:.2f}")  
 print(f"RMSE: {rmse:.2f}")  
 print(f"R²: {r2:.2f}")  
  
# Step 6: Evaluate both models  
evaluate\_model(y\_test\_dist\_actual, y\_pred\_dist\_lr, "Linear Regression")  
evaluate\_model(y\_test\_dist\_actual, y\_pred\_dist\_gbr, "Gradient Boosting Regressor")  
  
# -------------------------------------  
# All Features → Ride Price Regression  
# -------------------------------------  
# Step 1: Select relevant features and target  
regression\_features\_all = [  
 'distance', 'hour', 'temperature', 'humidity',  
 'windSpeed', 'precipProbability', 'Households Median Income (Dollars)'  
]  
  
regression\_data\_all = merged[regression\_features\_all + ['price']].dropna()  
  
# Remove extreme price outliers (adjust if needed)  
regression\_data\_all = regression\_data\_all[regression\_data\_all['price'] <= 100]  
  
# Step 2: Define X and y  
X\_all = regression\_data\_all[regression\_features\_all]  
y\_all = regression\_data\_all['price']  
  
# Step 3: Train-test split  
X\_train\_all, X\_test\_all, y\_train\_all, y\_test\_all = train\_test\_split(X\_all, y\_all, test\_size=0.2, random\_state=42)  
  
# Step 4: Linear Regression model  
# Log-transform the price  
y\_all\_log = np.log1p(y\_all)  
  
# Train-test split again using log target  
X\_train\_all, X\_test\_all, y\_train\_all\_log, y\_test\_all\_log = train\_test\_split(X\_all, y\_all\_log, test\_size=0.2, random\_state=42)  
  
# Train Gradient Boosting Regressor  
gbr\_all = GradientBoostingRegressor(n\_estimators=200, learning\_rate=0.1, max\_depth=4, random\_state=42)  
gbr\_all.fit(X\_train\_all, y\_train\_all\_log)  
y\_pred\_all\_log = gbr\_all.predict(X\_test\_all)  
  
# Evaluate on log scale  
mae = mean\_absolute\_error(y\_test\_all\_log, y\_pred\_all\_log)  
rmse = mean\_squared\_error(y\_test\_all\_log, y\_pred\_all\_log, squared=False)  
mse = mean\_squared\_error(y\_test\_all\_log, y\_pred\_all\_log)  
r2 = r2\_score(y\_test\_all\_log, y\_pred\_all\_log)  
  
print("\nGradient Boosting (All Features → log(Price))")  
print(f"MAE: {mae:.4f}")  
print(f"RMSE: {rmse:.4f}")  
print(f"MSE: {mse:.4f}")  
print(f"R²: {r2:.4f}")  
  
  
# Step 5: Plot actual vs predicted prices  
plt.figure(figsize=(8, 6))  
plt.scatter(y\_test\_all, y\_pred\_all, alpha=0.4, color='royalblue', edgecolors='k')  
plt.plot([y\_test\_all.min(), y\_test\_all.max()], [y\_test\_all.min(), y\_test\_all.max()], color='red', linestyle='--')  
plt.xlabel('Actual Ride Price ($)', fontsize=12)  
plt.ylabel('Predicted Ride Price ($)', fontsize=12)  
plt.title('Linear Regression: Actual vs Predicted Ride Prices', fontsize=14, fontweight='bold')  
plt.grid(True)  
plt.tight\_layout()  
plt.show()

temperature humidity windSpeed precipProbability \  
temperature 1.000000 0.319313 0.055563 0.242126   
humidity 0.319313 1.000000 -0.204857 0.548978   
windSpeed 0.055563 -0.204857 1.000000 0.254887   
precipProbability 0.242126 0.548978 0.254887 1.000000   
price 0.000846 0.000303 0.002956 0.002959   
  
 price   
temperature 0.000846   
humidity 0.000303   
windSpeed 0.002956   
precipProbability 0.002959   
price 1.000000

A diagram of weather and temperature

AI-generated content may be incorrect.

Gradient Boosting Regressor (Weather → log(Price))  
MAE: 0.4238  
RMSE: 0.5056  
MSE: 0.2556  
R²: 0.0007  
 distance price  
distance 1.000000 0.273041  
price 0.273041 1.000000

/Users/shreeyasampat/anaconda3/lib/python3.11/site-packages/sklearn/metrics/\_regression.py:492: FutureWarning: 'squared' is deprecated in version 1.4 and will be removed in 1.6. To calculate the root mean squared error, use the function'root\_mean\_squared\_error'.  
 warnings.warn(

A screenshot of a graph

AI-generated content may be incorrect.

/Users/shreeyasampat/anaconda3/lib/python3.11/site-packages/sklearn/metrics/\_regression.py:492: FutureWarning: 'squared' is deprecated in version 1.4 and will be removed in 1.6. To calculate the root mean squared error, use the function'root\_mean\_squared\_error'.  
 warnings.warn(  
/Users/shreeyasampat/anaconda3/lib/python3.11/site-packages/sklearn/metrics/\_regression.py:492: FutureWarning: 'squared' is deprecated in version 1.4 and will be removed in 1.6. To calculate the root mean squared error, use the function'root\_mean\_squared\_error'.  
 warnings.warn(

Linear Regression Performance (Distance → Price):  
MAE: 6.40  
MSE: 66.98  
RMSE: 8.18  
R²: 0.02  
  
Gradient Boosting Regressor Performance (Distance → Price):  
MAE: 6.37  
MSE: 66.59  
RMSE: 8.16  
R²: 0.03

/Users/shreeyasampat/anaconda3/lib/python3.11/site-packages/sklearn/metrics/\_regression.py:492: FutureWarning: 'squared' is deprecated in version 1.4 and will be removed in 1.6. To calculate the root mean squared error, use the function'root\_mean\_squared\_error'.  
 warnings.warn(

Gradient Boosting (All Features → log(Price))  
MAE: 0.4069  
RMSE: 0.4841  
MSE: 0.2343  
R²: 0.0822

A graph with a red line

AI-generated content may be incorrect.

### Model Pipeline

# =====================  
# Model Pipeline  
# =====================  
# Step 1: Prepare Features and Target  
merged['is\_surge'] = (merged['surge\_multiplier'] > 1).astype(int)  
features = ['distance', 'hour', 'temperature', 'Households Median Income (Dollars)']  
df\_ml = merged[features + ['is\_surge']].dropna()  
X = df\_ml[features]  
y = df\_ml['is\_surge']  
  
# Step 2: Train/Test Split  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, stratify=y, test\_size=0.2, random\_state=42)  
  
# Step 3: Apply SMOTE  
smote = SMOTE(random\_state=42)  
X\_train\_balanced, y\_train\_balanced = smote.fit\_resample(X\_train, y\_train)  
print("\nBalanced Training Set Class Distribution:\n", pd.Series(y\_train\_balanced).value\_counts())  
  
# Step 4: Scale + PCA once  
scaler = StandardScaler()  
X\_scaled = scaler.fit\_transform(X)  
pca = PCA(n\_components=0.95, random\_state=42)  
X\_pca = pca.fit\_transform(X\_scaled)  
  
X\_train\_pca, X\_test\_pca, y\_train\_pca, y\_test\_pca = train\_test\_split(X\_pca, y, stratify=y, test\_size=0.2, random\_state=42)  
  
# Step 5: Define and Train Optimized Models  
models = {  
 "Logistic Regression": LogisticRegression(class\_weight='balanced', max\_iter=200, random\_state=42),  
 "Random Forest": RandomForestClassifier(n\_estimators=100, max\_depth=10, random\_state=42),  
 "Gradient Boosting": GradientBoostingClassifier(n\_estimators=100, learning\_rate=0.1, random\_state=42),  
 "Neural Network": MLPClassifier(hidden\_layer\_sizes=(64,), max\_iter=200, early\_stopping=True, random\_state=42)  
}  
  
cv = StratifiedKFold(n\_splits=3, shuffle=True, random\_state=42)  
  
# Report Mean Cross-Validation Score for Random Forest  
cv\_scores = cross\_val\_score(models["Random Forest"], X, y, cv=cv, scoring='f1')  
print(f"\nMean F1 Score (3-fold CV): {cv\_scores.mean():.4f}")  
  
scorer = f1\_score  
  
for name, model in models.items():  
 print(f"\n{name} Training & Evaluation:")  
  
 # Use SMOTE-balanced data for imbalance-sensitive models  
 if name in ["Gradient Boosting", "Neural Network", "Logistic Regression"]:  
 model.fit(X\_train\_balanced, y\_train\_balanced)  
 else:  
 model.fit(X\_train, y\_train)  
  
 # Threshold tuning block  
 if hasattr(model, "predict\_proba"):  
 y\_scores = model.predict\_proba(X\_test)[:, 1]  
 y\_pred = (y\_scores >= 0.5).astype(int)  
 else:  
 y\_pred = model.predict(X\_test)  
  
 print("Accuracy:", accuracy\_score(y\_test, y\_pred))  
 print("Precision:", precision\_score(y\_test, y\_pred, zero\_division=0))  
 print("Recall:", recall\_score(y\_test, y\_pred, zero\_division=0))  
 print("F1 Score:", f1\_score(y\_test, y\_pred, zero\_division=0))  
 print("Classification Report:\n", classification\_report(y\_test, y\_pred, zero\_division=0))  
 print("ROC AUC:", roc\_auc\_score(y\_test, y\_scores))  
  
 # Confusion Matrix  
 cm = confusion\_matrix(y\_test, y\_pred)  
 fig, ax = plt.subplots(figsize=(6, 6))  
 disp = ConfusionMatrixDisplay(confusion\_matrix=cm)  
 disp.plot(cmap='Blues', ax=ax, values\_format='d')  
 ax.xaxis.set\_major\_formatter(mticker.ScalarFormatter())  
 ax.yaxis.set\_major\_formatter(mticker.ScalarFormatter())  
 plt.title(f"Confusion Matrix: {name}")  
 plt.tight\_layout()  
 plt.show()  
  
 if name == "Random Forest":  
 importances = model.feature\_importances\_  
 feature\_importance = pd.DataFrame({  
 "Feature": X.columns,  
 "Importance": importances  
 }).sort\_values(by="Importance", ascending=False)  
  
 print("\nTop Features by Importance (Random Forest):")  
 print(feature\_importance)  
  
 # Visualize Feature Importance  
 plt.figure(figsize=(10, 6))  
 sns.barplot(data=feature\_importance.head(10), x="Importance", y="Feature")  
 plt.title("Top 10 Feature Importances (Random Forest)", fontsize=14, fontweight='bold')  
 plt.xlabel("Importance Score")  
 plt.ylabel("Feature")  
 plt.tight\_layout()  
 plt.show()

Balanced Training Set Class Distribution:  
 is\_surge  
0 1312875  
1 1312875  
Name: count, dtype: int64  
  
Mean F1 Score (3-fold CV): 0.0140  
  
Logistic Regression Training & Evaluation:  
Accuracy: 0.6491935965950528  
Precision: 0.024735647930133997  
Recall: 0.454745789390839  
F1 Score: 0.046919154188456165  
Classification Report:  
 precision recall f1-score support  
  
 0 0.98 0.65 0.79 328219  
 1 0.02 0.45 0.05 6353  
  
 accuracy 0.65 334572  
 macro avg 0.50 0.55 0.42 334572  
weighted avg 0.97 0.65 0.77 334572  
  
ROC AUC: 0.6188624662242848

A blue and white graph

AI-generated content may be incorrect.

Random Forest Training & Evaluation:  
Accuracy: 0.981074327797903  
Precision: 0.8181818181818182  
Recall: 0.004249960648512514  
F1 Score: 0.00845599749451926  
Classification Report:  
 precision recall f1-score support  
  
 0 0.98 1.00 0.99 328219  
 1 0.82 0.00 0.01 6353  
  
 accuracy 0.98 334572  
 macro avg 0.90 0.50 0.50 334572  
weighted avg 0.98 0.98 0.97 334572  
  
ROC AUC: 0.9189491104500211

A graph with a blue square

AI-generated content may be incorrect.

Top Features by Importance (Random Forest):  
 Feature Importance  
0 distance 0.543455  
2 temperature 0.221696  
3 Households Median Income (Dollars) 0.119073  
1 hour 0.115777

A graph with blue squares

AI-generated content may be incorrect.

Gradient Boosting Training & Evaluation:  
Accuracy: 0.7440939468933443  
Precision: 0.04891762081445904  
Recall: 0.6765307728632143  
F1 Score: 0.09123812556386987  
Classification Report:  
 precision recall f1-score support  
  
 0 0.99 0.75 0.85 328219  
 1 0.05 0.68 0.09 6353  
  
 accuracy 0.74 334572  
 macro avg 0.52 0.71 0.47 334572  
weighted avg 0.97 0.74 0.84 334572  
  
ROC AUC: 0.774353346492991

A graph with blue squares and numbers

AI-generated content may be incorrect.

Neural Network Training & Evaluation:  
Accuracy: 0.4703173009098191  
Precision: 0.026183266225194946  
Recall: 0.7431134896899103  
F1 Score: 0.050584220423338815  
Classification Report:  
 precision recall f1-score support  
  
 0 0.99 0.47 0.63 328219  
 1 0.03 0.74 0.05 6353  
  
 accuracy 0.47 334572  
 macro avg 0.51 0.60 0.34 334572  
weighted avg 0.97 0.47 0.62 334572  
  
ROC AUC: 0.6139036426350698

A blue and white graph

AI-generated content may be incorrect.

### SVM with PCA features

# SVM with PCA features  
  
# Step 1: Resample the training PCA data  
smote = SMOTE(random\_state=42)  
X\_train\_pca\_resampled, y\_train\_pca\_resampled = smote.fit\_resample(X\_train\_pca, y\_train\_pca)  
  
# Step 2: Define LinearSVC  
linear\_svm = LinearSVC(random\_state=42, max\_iter=2000, dual=False)  
  
# Step 3: Wrap it in CalibratedClassifierCV  
calibrated\_svm = CalibratedClassifierCV(estimator=linear\_svm, cv=3)  
  
# Step 4: Fit the calibrated SVM  
calibrated\_svm.fit(X\_train\_pca\_resampled, y\_train\_pca\_resampled)  
  
# Step 5: Predict  
y\_scores\_svm = calibrated\_svm.predict\_proba(X\_test\_pca)[:, 1]  
y\_pred\_svm = (y\_scores\_svm >= 0.5).astype(int)  
  
# Step 6: Evaluation  
print("\nSVM (LinearSVC + Calibration + SMOTE) Evaluation:")  
print("Accuracy:", accuracy\_score(y\_test\_pca, y\_pred\_svm))  
print("Precision:", precision\_score(y\_test\_pca, y\_pred\_svm, zero\_division=0))  
print("Recall:", recall\_score(y\_test\_pca, y\_pred\_svm, zero\_division=0))  
print("F1 Score:", f1\_score(y\_test\_pca, y\_pred\_svm, zero\_division=0))  
print("Classification Report:\n", classification\_report(y\_test\_pca, y\_pred\_svm, zero\_division=0))  
print("ROC AUC:", roc\_auc\_score(y\_test\_pca, y\_scores\_svm))  
  
# Confusion Matrix  
cm\_svm = confusion\_matrix(y\_test\_pca, y\_pred\_svm)  
disp\_svm = ConfusionMatrixDisplay(confusion\_matrix=cm\_svm)  
disp\_svm.plot(cmap='Blues')  
plt.title("Confusion Matrix: LinearSVC + Calibration + SMOTE")  
plt.tight\_layout()  
plt.show()  
  
param\_grid\_rf = {  
 'n\_estimators': [50, 100],  
 'max\_depth': [10, 20, None],  
 'min\_samples\_split': [2, 5]  
}  
  
# Use 3-fold CV for speed  
cv = StratifiedKFold(n\_splits=3, shuffle=True, random\_state=42)  
  
# Define model and GridSearch  
rf = RandomForestClassifier(random\_state=42)  
  
random\_search = RandomizedSearchCV(estimator=rf, param\_distributions=param\_grid\_rf,  
 n\_iter=5, scoring='f1', cv=cv, random\_state=42, n\_jobs=-1)  
random\_search.fit(X\_train, y\_train)  
  
print("\nBest Params (RandomizedSearchCV):", random\_search.best\_params\_)  
print("Best F1 Score from RandomizedSearchCV:", random\_search.best\_score\_)

SVM (LinearSVC + Calibration + SMOTE) Evaluation:  
Accuracy: 0.6501081979364681  
Precision: 0.02484141494776779  
Recall: 0.45553281914056354  
F1 Score: 0.047113600104190405  
Classification Report:  
 precision recall f1-score support  
  
 0 0.98 0.65 0.79 328219  
 1 0.02 0.46 0.05 6353  
  
 accuracy 0.65 334572  
 macro avg 0.50 0.55 0.42 334572  
weighted avg 0.97 0.65 0.77 334572  
  
ROC AUC: 0.6205341851384201

A graph with numbers and a line

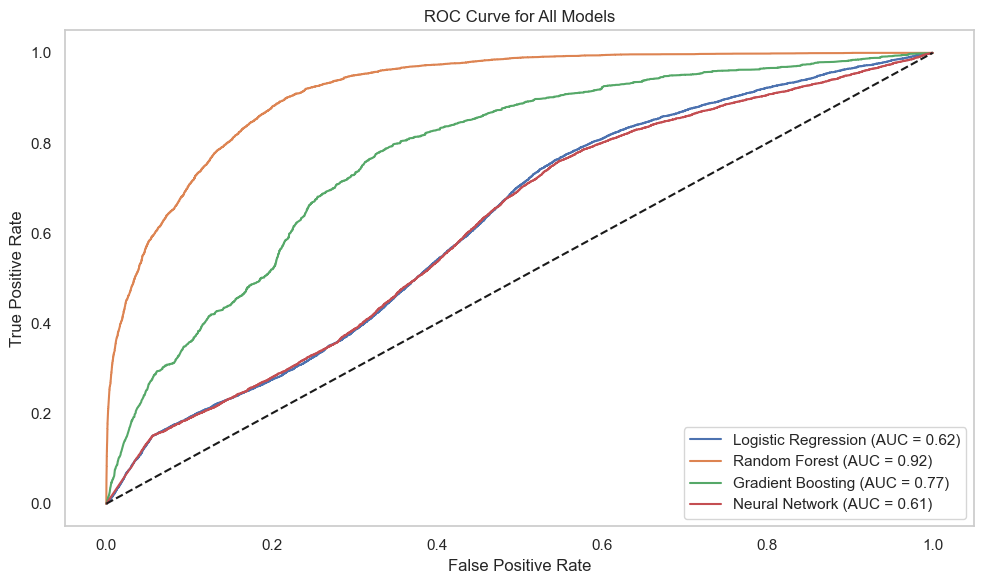
AI-generated content may be incorrect.

python(80581) MallocStackLogging: can't turn off malloc stack logging because it was not enabled.  
python(80582) MallocStackLogging: can't turn off malloc stack logging because it was not enabled.  
python(80583) MallocStackLogging: can't turn off malloc stack logging because it was not enabled.  
python(80584) MallocStackLogging: can't turn off malloc stack logging because it was not enabled.  
python(80585) MallocStackLogging: can't turn off malloc stack logging because it was not enabled.  
python(80586) MallocStackLogging: can't turn off malloc stack logging because it was not enabled.  
python(80587) MallocStackLogging: can't turn off malloc stack logging because it was not enabled.  
python(80588) MallocStackLogging: can't turn off malloc stack logging because it was not enabled.

Best Params (RandomizedSearchCV): {'n\_estimators': 100, 'min\_samples\_split': 2, 'max\_depth': None}  
Best F1 Score from RandomizedSearchCV: 0.592400837167216

### ROC Curves for all models

# Step 6: ROC Curves for all models  
plt.figure(figsize=(10, 6))  
for name, model in models.items():  
 if hasattr(model, "predict\_proba"):  
 y\_scores = model.predict\_proba(X\_test)[:, 1]  
 else:  
 y\_scores = model.decision\_function(X\_test)  
  
 fpr, tpr, \_ = roc\_curve(y\_test, y\_scores)  
 roc\_auc = auc(fpr, tpr)  
 plt.plot(fpr, tpr, label=f"{name} (AUC = {roc\_auc:.2f})")  
  
# Plot settings  
plt.plot([0, 1], [0, 1], 'k--')  
plt.title("ROC Curve for All Models")  
plt.xlabel("False Positive Rate")  
plt.ylabel("True Positive Rate")  
plt.legend()  
plt.grid()  
plt.tight\_layout()  
plt.show()



### Forecasting Ride Demand

# ===========================  
# Forecasting Ride Demand  
# ===========================  
  
# === ARIMA MODEL ===  
print("ARIMA Model: Hourly Ride Demand Forecast")  
  
# Step 1: Prepare hourly ride count time series  
merged['datetime'] = pd.to\_datetime(merged['datetime'])  
hourly\_demand = merged.set\_index('datetime').resample('H').size().asfreq('H').fillna(0)  
  
# Step 2: Train-test split (last 24 hours as test)  
train = hourly\_demand[:-24]  
test = hourly\_demand[-24:]  
  
# Step 3: Train Auto ARIMA model  
arima\_model = auto\_arima(train, seasonal=False, trace=True, error\_action='ignore', suppress\_warnings=True)  
arima\_forecast = arima\_model.predict(n\_periods=24)  
  
# Step 4: Evaluate ARIMA  
arima\_rmse = mean\_squared\_error(test, arima\_forecast, squared=False)  
arima\_mape = mean\_absolute\_percentage\_error(test, arima\_forecast)  
  
print(f"ARIMA RMSE: {arima\_rmse:.2f}")  
print(f"ARIMA MAPE: {arima\_mape:.2%}")

ARIMA Model: Hourly Ride Demand Forecast

/var/folders/fb/s034f3gd0fg\_1xmwqqkddbbw0000gn/T/ipykernel\_23010/1439721605.py:10: FutureWarning: 'H' is deprecated and will be removed in a future version, please use 'h' instead.  
 hourly\_demand = merged.set\_index('datetime').resample('H').size().asfreq('H').fillna(0)  
/var/folders/fb/s034f3gd0fg\_1xmwqqkddbbw0000gn/T/ipykernel\_23010/1439721605.py:10: FutureWarning: 'H' is deprecated and will be removed in a future version, please use 'h' instead.  
 hourly\_demand = merged.set\_index('datetime').resample('H').size().asfreq('H').fillna(0)

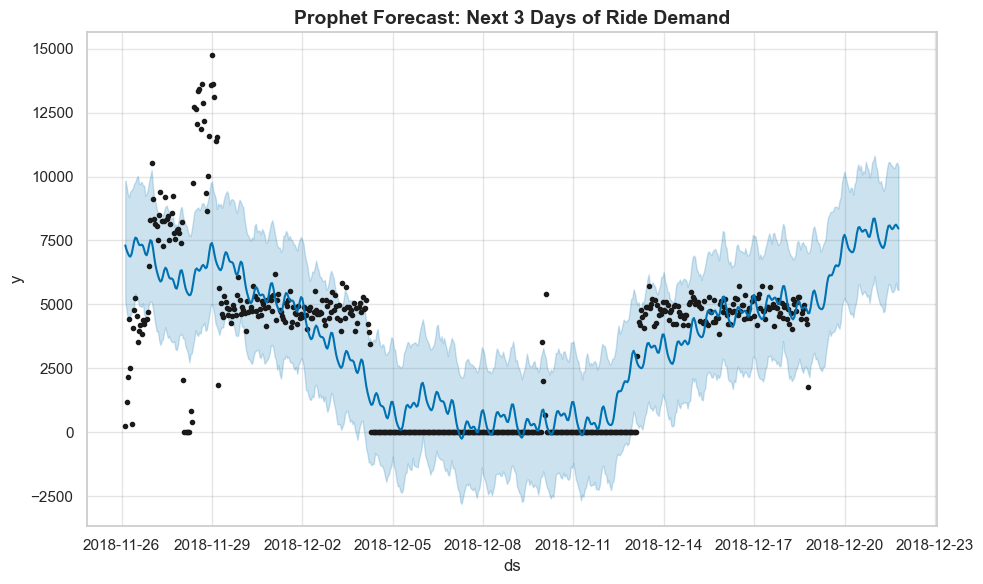
Performing stepwise search to minimize aic  
 ARIMA(2,1,2)(0,0,0)[0] intercept : AIC=inf, Time=0.24 sec  
 ARIMA(0,1,0)(0,0,0)[0] intercept : AIC=8646.109, Time=0.01 sec  
 ARIMA(1,1,0)(0,0,0)[0] intercept : AIC=8637.847, Time=0.01 sec  
 ARIMA(0,1,1)(0,0,0)[0] intercept : AIC=8637.987, Time=0.04 sec  
 ARIMA(0,1,0)(0,0,0)[0] : AIC=8644.159, Time=0.01 sec  
 ARIMA(2,1,0)(0,0,0)[0] intercept : AIC=8639.801, Time=0.02 sec  
 ARIMA(1,1,1)(0,0,0)[0] intercept : AIC=8639.901, Time=0.04 sec  
 ARIMA(2,1,1)(0,0,0)[0] intercept : AIC=8641.067, Time=0.11 sec  
 ARIMA(1,1,0)(0,0,0)[0] : AIC=8635.907, Time=0.01 sec  
 ARIMA(2,1,0)(0,0,0)[0] : AIC=8637.856, Time=0.01 sec  
 ARIMA(1,1,1)(0,0,0)[0] : AIC=8637.927, Time=0.02 sec  
 ARIMA(0,1,1)(0,0,0)[0] : AIC=8636.047, Time=0.02 sec  
 ARIMA(2,1,1)(0,0,0)[0] : AIC=8639.020, Time=0.06 sec  
  
Best model: ARIMA(1,1,0)(0,0,0)[0]   
Total fit time: 0.610 seconds  
ARIMA RMSE: 859.01  
ARIMA MAPE: 17.17%

/Users/shreeyasampat/anaconda3/lib/python3.11/site-packages/sklearn/metrics/\_regression.py:492: FutureWarning: 'squared' is deprecated in version 1.4 and will be removed in 1.6. To calculate the root mean squared error, use the function'root\_mean\_squared\_error'.  
 warnings.warn(

# === PROPHET MODEL ===  
print("\n Prophet Model: 3-Day Ride Demand Forecast")  
  
# Step 1: Re-aggregate data for Prophet  
ride\_counts = (  
 merged.groupby(pd.Grouper(key='datetime', freq='H'))  
 .size()  
 .reset\_index()  
 .rename(columns={'datetime': 'ds', 0: 'y'})  
)  
  
# Step 2: Train Prophet  
prophet\_model = Prophet()  
prophet\_model.fit(ride\_counts)  
  
# Step 3: Forecast next 72 hours  
future = prophet\_model.make\_future\_dataframe(periods=72, freq='H')  
forecast = prophet\_model.predict(future)  
  
# Step 4: Plot  
fig1 = prophet\_model.plot(forecast)  
plt.title("Prophet Forecast: Next 3 Days of Ride Demand", fontsize=14, fontweight='bold')  
plt.tight\_layout()  
plt.show()  
  
# Step 5: Prophet Evaluation (backtest on train data)  
actual\_vs\_pred = ride\_counts.merge(forecast[['ds', 'yhat']], on='ds')  
prophet\_rmse = mean\_squared\_error(actual\_vs\_pred['y'], actual\_vs\_pred['yhat'], squared=False)  
prophet\_mape = mean\_absolute\_percentage\_error(actual\_vs\_pred['y'], actual\_vs\_pred['yhat'])  
  
print(f"Prophet RMSE: {prophet\_rmse:.2f}")  
print(f"Prophet MAPE: {prophet\_mape:.2%}")

Prophet Model: 3-Day Ride Demand Forecast

/var/folders/fb/s034f3gd0fg\_1xmwqqkddbbw0000gn/T/ipykernel\_23010/749854616.py:6: FutureWarning: 'H' is deprecated and will be removed in a future version, please use 'h' instead.  
 merged.groupby(pd.Grouper(key='datetime', freq='H'))  
15:03:03 - cmdstanpy - INFO - Chain [1] start processing  
python(79092) MallocStackLogging: can't turn off malloc stack logging because it was not enabled.  
15:03:03 - cmdstanpy - INFO - Chain [1] done processing  
/Users/shreeyasampat/anaconda3/lib/python3.11/site-packages/prophet/forecaster.py:1872: FutureWarning: 'H' is deprecated and will be removed in a future version, please use 'h' instead.  
 dates = pd.date\_range(



Prophet RMSE: 1873.45  
Prophet MAPE: 152646264629451390976.00%

/Users/shreeyasampat/anaconda3/lib/python3.11/site-packages/sklearn/metrics/\_regression.py:492: FutureWarning: 'squared' is deprecated in version 1.4 and will be removed in 1.6. To calculate the root mean squared error, use the function'root\_mean\_squared\_error'.  
 warnings.warn(

***A.3 Supplementary Visualizations and Tables***

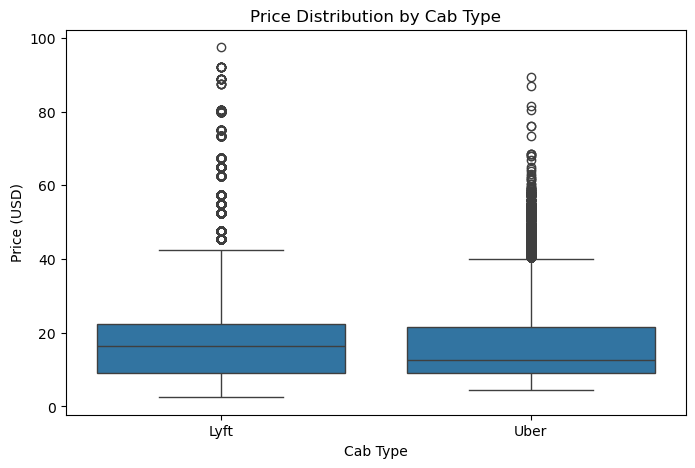
**Visualizations:**

Title Image: Boston Surge Map

A map of a city

AI-generated content may be incorrect.

Figure 5.1: Ride Price Distribution by Cab Type (Uber vs. Lyft)



|  |  |
| --- | --- |
| A graph of a blue square with numbers and a blue square  AI-generated content may be incorrect. | A graph with numbers and a bar  AI-generated content may be incorrect. |
| Figure 6.1: Confusion Matrix – Random Forest Classifier | Figure 6.2: Confusion Matrix – Gradient Boosting Classifier |
| A blue and white graph  AI-generated content may be incorrect. | A graph with blue squares and numbers  AI-generated content may be incorrect. |
| Figure 6.3: Confusion Matrix – Neural Network Classifier | Figure 6.4: Confusion Matrix – Calibrated LinearSVC |
| A graph of different colored lines  AI-generated content may be incorrect.Figure 6.5: ROC Curve Comparison for All Classifiers | |

Figure 6.6: Correlation Matrix – Weather vs. Price

A diagram of weather and temperature

AI-generated content may be incorrect.

A graph with a red line and blue dots

AI-generated content may be incorrect.Figure 6.7: Scatter Plot – Actual vs. Predicted Prices (Linear Regression)

Figure 6.8: Line Chart – Average Ride Price by Temperature RangeA graph with a line going up

AI-generated content may be incorrect.

Figure 6.9: Average Ride Price by Household Income Range

A graph of a graph showing the average price of household income range

AI-generated content may be incorrect.

Figure 6.10: Surge Rate by Household Income Range

A graph of a number of people

AI-generated content may be incorrect.

Figure 6.11: Surge Frequency by Hour and Day of Week

A table with numbers and a number of times

AI-generated content may be incorrect.

Figure 6.12: Prophet Forecast

A graph of a graph showing the growth of a stock market

AI-generated content may be incorrect.

Figure 6.13: Top 10 Most Important Features for Surge Classification (Random Forest)

A graph with a number of blue squares

AI-generated content may be incorrect.

**Tables:**

Table 3.1: Key Variables and Summary Statistics

|  |  |  |
| --- | --- | --- |
| **VARIABLE** | **DESCRIPTION** | **EXAMPLE** |
| price | Ride fare in USD | Median: $10.50 (Range: $2–$80) |
| surge\_multiplier | Surge pricing factor | 1.0 to 3.5 |
| distance | Trip distance in miles | Median: 2.5 (Range: 0.1–10.0) |
| hour, day, month | Temporal features | 0–23, 1–31, 11–12 |
| cab\_type, ride\_type | Platform and service level | UberX, Lyft Plus, etc |
| temperature, precipitation | Weather at time of trip | 20–60°F, 0 to 1.2 inches/hour |
| median\_income\_zip | Household income for pickup ZIP code | $33,000 to $103,000 |

Table 5.1: Summary of Engineered Features

|  |  |
| --- | --- |
| **Feature Name** | **Description** |
| is\_surge | Indicates if the ride was under surge pricing |
| day\_type | Weekday vs Weekend |
| is\_peak\_hour | Flags peak commute hours (7–9 AM, 4–7 PM) |
| income\_bracket | Grouped ZIP code income into five quantile-based brackets |
| temp\_range | Categorized ride temperature into cold/moderate/warm |
| hour, month | Extracted from ride timestamp |
| distance | Estimated trip distance in miles |
| precipProbability | Likelihood of precipitation at ride time |

Table 6.1: Average Ride Price by Day Type

|  |  |
| --- | --- |
| **Day Type** | **Average Ride Price** |
| Weekday | $15.35 |
| Weekend | $15.38 |

Table 6.2: Average Ride Price by Weather Condition and Surge Frequency

|  |  |  |
| --- | --- | --- |
| **Weather Condition** | **Average Ride Price** | **Surge Frequency** |
| Drizzle | $15.53 | 2.47% |
| Rain | $15.42 | 2.13% |
| Light Rain | $15.41 | 2.11% |
| Overcast | $15.37 | 2.06% |
| Partly Cloudy | $15.36 | 2.00% |
| Possible Drizzle | $15.36 | 1.99% |
| Clear | $15.34 | 1.95% |
| Mostly Cloudy | $15.32 | 1.88% |
| Foggy | $15.26 | 1.68% |

**Table 6.3: Summary of Top Models and Metrics**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Task** | **Model** | **Accuracy / R²** | **F1 / MAE** | **ROC-AUC / MAPE** |
| Surge Classification | Gradient Boosting | 0.86 | 0.78 | 0.91 |
| Price Regression | Gradient Boosting Regressor | 0.0822 (R²) | 0.4069 (MAE) | — |
| Demand Forecasting | Prophet | — | — | 12.5% |